A Study of Norms and Standards regarding Embedded Decisional Software

Philippe Morignot, Jean-François Tilman, Christophe Cognault
AXLOG Ingénierie, 19-21 rue du 8 mai 1945, 94110 Arcueil, France.
Phone: +33 (0)1 41 24 31 19       Fax: +33 (0)1 41 24 07 36
{Philippe.Morignot; Jean-Francois.Tilman; Christophe.Cognault}@axlog.fr

Pierre Hélie, Bruno Patin
DASSAULT AVIATION, 33 rue Ferdinand Forrest, 92510 Suresnes, France.
Phone: +33 (0)1 47 11 58 54       Fax: +33 (0)1 47 11 53 65

Keywords embedded decisional software, norms and standards, vehicle system modeling.

Abstract

A piece of software is critical when the consequences of its malfunction can damage materials, injure or even kill Humans. This type of software usually requires 5 or 6 times the development time of non-critical software. To control the safety of such critical software, many norms and standards have been proposed (e.g., in Europe, in the USA, in specific countries). These procedures help classifying and analyzing these risks, and finally proposing solutions to reduce them to an acceptable level.

Now let us consider the case of decisional software: these pieces of software take decisions on the course of actions of an aircraft, a spacecraft, or, more generally, of any autonomous system, instead of having a Human deciding on them. Hence these pieces of software have potentially a high impact on the safety of a system or device (a wrong decision may result in the crash of the system, potentially with consequences for human beings), such as uninhabited aerial vehicles (UAV).

In this paper, we present a study of norms and standards from the point of view of decisional software embedded in UAV or autonomous spacecraft. On the one hand, we analyze the existing norms and standards in (aero-) space, safety and software domains, to exhibit how they constrain the design of decisional software (e.g., regarding non determinism, verifiability, memory space, response time). On the other hand, we summarize the characteristics of decisional software (e.g., non determinism, combinatorial explosion resulting in huge memory space and too long response time). Based on these, we propose recommendations, inspired by the domain or Artificial Intelligence, to adapt norms and standards to this kind of software (e.g., study of the shape of a search space, semantic phases in the input variables, taxonomy of tests, use of anytime algorithms, software architecture of agents, limited depth in the search tree).

1. Introduction

Embedding decisional software leads to autonomous aircraft: they can fly without the need for a human pilot. But embedding this kind of software is restricted by norms

1 This work has been funded by the Délégation Générale pour l'Armement (the French Procurement Agency).
and standards, that force the development team to analyze the risks of such a project. When decisional functions are at stake, such risks are high, since they control the aircraft: if these decisional functions go wrong, there could be Human injuries or death. This stresses the importance of confronting existing norms and standards, accumulated over the years for good software development, to the needs of software implementing decisional functions.

The paper is organized as follows: first, we present an overview of the applicable standards and norms, with a brief summary of the main points. Then we sum up the characteristics of decisional software. In a third section, we match these standards and norms against decisional software requirements, and see where the difficult points are. We propose recommendations for fixing standards and norms regarding decisional software. Finally we sum up our contribution.

2. Standards and norms

When considering norms and standards regarding embedded decisional software, the following domains of norms are considered: aerial, spatial, software, safety. All norms define conditions under which some system (not necessarily software) can be certified --- once a piece of software has passed certification, it can be used in the designated environment.

Aerial: Since its origin in 1980, the norm DO-178B (in the USA) [1] / ED-12 EUROCAE (in Europe) [2] builds categories of failures:

- **Catastrophic**: failure conditions that can prevent the safe unfolding of a flight, or the capability of the crew to face hostile conditions going even to (1) an important reduction of safety range or functional capability, (2) physical problems or workload increase so that the crew cannot accomplish its tasks in a precise or complete way, (3) negative effects on passengers such as important injuries or even deaths.
- **Dangerous**: failure conditions that can reduce the capability of the aircraft/crew to face hostile conditions, e.g., important reduction of safety range or functional capacity; physical problems or load increase such that the work of the crew cannot be performed normally; negative effective on the passengers such as injuries or death.
- **Major**: close to the dangerous, except that the reduction of safety range is high (instead of important), the crew can keep ensuring its work but with difficulty only, the passengers can suffer from discomfort, eventually injuries.
- **Minor**: failure conditions not leading to significant reduction of the aircraft safety.
- **Without effect**: failure conditions not changing the operational capability of the aircraft or the workload of the crew.

To each failure conditions is associated a software level. Some techniques are suggested to reach them: (i) partitioning ensures the isolation of software components (hence, of their failures); (ii) a piece of software is encoded with different multiple versions, hoping that failures may be different in each case; (iii) surveillance of a piece of software, i.e., checking the outputs of a function.

This norm also advises processes for software checking, software configuration management, software quality insurance, coordination towards certification. It has been refined in its version C regarding object-oriented languages [2].
The Society of Automotive Engineers issued the standards ARP 4754 and 4761 [3] in 1996, which are compatible with the norm DO-178B and are used in aircraft safety domains.

Safety: The standard MIL-STD-882 [7] deals with the safety of systems in its different versions since 1969 --- it is not limited to software. It provides uniform requirements for analyzing system danger, and imposes requirements to control, eliminate or reduce these risks. It regulates the exchange between a managing activity and a contractor. As for DO-178B, it defines mishaps, hazards, danger severity and risk. A system safety program must be established, analyzing the occurrence of these concepts in the system under development. This standard recommends incorporating this analysis and consequences in the design phase. If danger cannot be eliminated, the system must be re-designed. If this is not possible, additional safety devices must be set. If this is not possible, fault detection and warning must be set. If this is still not possible, user training must be set, in order to face unavoidable danger. 22 tasks are defined in four sections: program management and control, design and integration, design assessment, compliance and verification.

Another English standard, Def Stan 00-56 [8] in 1996, provides uniform requirements to implement an analysis of the safety of a system --- it is not limited to software either. The defined concepts are accident severity (catastrophic, critical, marginal, negligible), accident probability (frequent, probable, occasional, remote, improbable, incredible), and risks (intolerable, undesired, tolerable if accepted by the safety committee, tolerable). On all risk classes, a preliminary hazard identification, preliminary hazard analysis and a system change hazard analysis must be performed. On the highest classes of risks, a system hazard analysis and a system risk assessment must be performed. A safety integrity level defines an indicator (S1 to S4) of the required level of protection against systematic failures (see Figure 1).

<table>
<thead>
<tr>
<th>Failure probability of first function</th>
<th>Accident severity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequent</td>
<td>Catastrophic</td>
</tr>
<tr>
<td>Probable</td>
<td>Level S4</td>
</tr>
<tr>
<td>Occasional</td>
<td>Critical</td>
</tr>
<tr>
<td>Remote</td>
<td>Level S3</td>
</tr>
<tr>
<td>Improbable</td>
<td>Marginal</td>
</tr>
<tr>
<td></td>
<td>Level S2</td>
</tr>
<tr>
<td></td>
<td>Negligible</td>
</tr>
<tr>
<td></td>
<td>Level S1</td>
</tr>
</tbody>
</table>

Figure 1: Safety integrity levels.

A hazard log must be maintained throughout each danger analysis, and this log is synthesized into a safety case. Recommendations to lower the safety integrity level are: re-specifications, re-design, addition of safety devices, addition of warning devices, operational procedures and user training. Potential accidents are sorted into a functional analysis (is this danger due to a correct or incorrect functioning of the system?), zonal analysis (what are the consequences of failures on other systems?), failure analysis (analysis of the failure modes and their consequences), risk analysis (associating danger to scenarios).

Software: The norm, MIL-STD-498 [4] (its commercial version is J-STD-016), has evolved since 1994 to become IEEE/EIA 12207 in 1998. It covers software development for military applications. It applies to software cycle development and regulates the exchanges between an acquirer and a developer. It consists in general (through data item descriptions, e.g., development plan, documented methods, use of other standards, software reuse, strategies for critical requirements, hardware constraints, justifications of key decisions) and specific (computer software configuration items, e.g., CSCI planning, development environment, system architectural design, software design, implementation and integration units) requirements.

applies to system and software acquisition, providing, development, exploitation and maintenance of software. It is composed of basic (acquisition, providing, development, exploitation, maintenance), support (documentation, configuration management, quality insurance, verification, validation, joint review, audit, problem solving) and organizational (management, infrastructure, process improvement, education) processes. Each process is composed of activities, in turn composed of tasks.

The English standard Def Stan 00-55 [9], edited in 1997, provides requirements for safety-related software in defense equipment. It is based on the standard Def Stan 00-56. A first requirement is to analyze the software through Def Stan 00-56 to establish the safety integrity levels: S1, S2, S3 lead to safety related software, S4 leads to safety critical software. Various processes propose to produce and maintain a software safety plan, produce and maintain a software safety case, accumulate the history of the analysis into a software safety records log and other items. Other standards must be used in conjunction with Def Stan 00-55: the quality system must be certified with ISO 9901, the application guides with ISO 9000-3, documents must be written in accordance with Def Stan 05-91, configuration management with Def Stan 05-57. Software related activities include formal specification and design of software, structured design method, static analysis of the source code (source code coverage, metrics, information flow, semantic analysis, etc), and dynamic tests. A high level programming language must be used. A language must be strongly typed, structured by block, have a formally defined syntax and be deterministic. Code generation must be used as much as possible --- with the generation chain certified first. Formal proofs must be given regarding the internal consistency of a piece of software. Modularity, encapsulation, abstraction, fault tolerance, predictability, analysis of the object code, etc, is recommended. Programming techniques such as concurrence, interruption, recursion, floating point must be avoided.

Spatial: The standard ECSS-E-40 Part 1B [10], led by ESA, is related to spatial applications in Europe. ECSS is declined into management (-M), engineering (-E) and insurance product (-Q), in addition to general documents (-P). ECSS-E-40 deals with the structure and the content of documents, based on a model of client/provider, repeated on all levels of the life cycle of the software. The processes are software management, system engineering, requirement and architectural engineering, design and development, software validation, software delivery and acceptance, software verification, software maintenance.

3. Decisional software

A piece of software is said to be decisional when it takes some decision at some point in time. By “decision”, we mean that some alternatives are open, and the software actually decides to follow one instead of the others. This leads in turn to other such choice points, which lead to further decisions, etc. In other words, a tree of possibilities is unfolded and a piece of decisional software has to find one solution inside it. By “solution”, we mean a state (i.e., a node in this tree, where arcs are decisions) where some property holds --- this property actually defines what a solution is.

This whole process is called a search in some state space. Looking for a solution turns out to unfold this tree and scanning it for a state with the above property. Now, the size of the tree can be shown to be exponential: the number of states/nodes to scan increases exponentially with the depth of the tree. Therefore, even for small
problems, the number of states to scan can be larger than the number of molecules in the Universe --- combinatorial explosion.

Many algorithms exist to scan this search tree, depending on the problem at hand and on the shape of the search space. For example, some may use heuristics (i.e., other information to help choosing among the branches of this tree), in order to look for the best estimated state (best-first search), or the best estimated sequence of states (A*). Others may go deep inside the tree, but can propagate the consequences of their choices, hence detecting impossibilities, which may lead them to go backward and look for solutions in other branches (constraint satisfaction problems). Others simply are uninformed (no heuristics) and blindly follow a pattern of tree expansion (depth-first search, breadth-first search). Others use probabilistic models (re-inforcement learning, hidden Markov models) to represent the capability of an algorithm to learn in advance the correct choice, which results in a very fast response at run time -- a distinction should be made between a learning phase and the production phase.

The problems for which a tree of states has to be developed, in order for an algorithm to find a solution, belong to the category of NP-hard problems [6] --- “NP” standing for “non-deterministic polynomial”, which means that these problems have a polynomial complexity if an oracle/heuristics is given (and an exponential complexity otherwise).

4. Problems

We browse the numerous problems that decisional software encounter when embedded in an aerial vehicle for which norms and standards apply.

4.1. Non-determinism

Determinism is the property of an algorithm to always produce the same outputs given the same inputs. The previous algorithms are non deterministic since they depend on the heuristics given to scan the search tree. But once these heuristic have been chosen, the only source of non determinism inside such an algorithm is random numbers.

Norms and standards usually do not require some source code to be deterministic (DO-178B, Def Stan 00-55, ECSS-E-40). These non deterministic algorithms (e.g., simulated annealing) do not usually take part into source code developed for certification.

4.2. Verifiability

The DO-178B norm, for example, stresses that source code must be verifiable. This property encompasses tests, code review and code analysis, to check that some source code satisfies its specifications.

Tests: A first test is to prove termination, correctness and completion. The difficulty comes from:

- All outputs of a decisional function cannot be tested, since there are simply too many of them.
- Even on simple cases, a decisional function may return a solution which is not the expected one: the function has found another solution in the search tree, which may be close to or far away from the expected solution. And this is perfectly normal: the density of solutions may highly vary from one problem to another, from one data set to another.

Structural coverage of tests: structural tests rely on the structure of the source code of some function. They aim at
covering all instructions, all branches, all decision points and make explicit the conditions under which these instructions, branches and decision points are under the control flow of the computer. A consequence of these tests is to detect dead source code --- the control flow will never pass on them. The difficulty comes from:

- The computer languages used for encoding decisional source code may be far from imperative computer languages considered in norms and standards. Languages such as Prolog highly differ from Ada or C. The structure of such programs highly differs from the one currently used in software that attempts at being certified.
- Structural tests, although required, are not pertinent for decisional functions. The complexity of a decisional function does not reside in its source code, but rather in the search tree that it explores. This tree has to be tested, testing the source code of a decisional function is not representative.
- Decisional functions usually rely on existing software modules (e.g., solvers, interpreters). The source code of these modules must be tested too, according to these norms and standards, which is not always possible --- companies do not provide the source code of a Prolog interpreter to their customers.

On complex cases, verifiability becomes more difficult: even a Human does not have a solution, any solution provided by a computer is accepted since it is better than the absence of solution that a Human can provide. An assessment of the consistency of this solution is still required by a Human expert.

5. Solutions

5.1. Introduction

Norms and standards almost always impose an analysis grid of danger and risks. Now if we consider the case of decisional software embedded into aircraft (software controlling the aircraft), it must be acknowledged that all accidents are possible due to software failure.

Many accidents can then intervene: loss of control of the aircraft, leading in a crash on the ground, a collision against another aircraft, or even an unexpected drop of weapon (in the case of military aircraft autonomously driven). Without any other controls, such a software failure could lead to death of severe injury of humans --- either civil or military. Therefore, the accident is catastrophic, according to the previous norms and standards taxonomies. We present into more details the recommendations that we make to lower the safety integrity level.

5.2. Black box

A first step is to detect failures or exhausted threshold. As for non decisional software, several measures are possible:

- **auto-tests**: keeping sending data to decisional functions and testing their output;
- **integrity tests**: same as auto-tests, but on demand, not continuously;
- **redundancy**: according to DO-178B, this consists in duplicating an uneven number of times a decisional function: the highest number of identical results wins;
- **domain tests**: setting a range for input and output data;
- **behavior surveillance**: encoding a diminished version of a decisional function, that performs fast surveillance on the real decisional function.

Once a failure is detected, a first reaction, is backup: a motion that changes the
trajectory of the aircraft, and that saves it, at least temporarily. For example, when an aircraft is about to crash on the ground, a backup function makes it go up. Redundancy leads to keep going with the mission of the aircraft with the same performances. Degraded performances may also be tolerated.

5.3. Glass box

Regarding non determinism, a first solution is to prevent the developer from using them. This may lead to prevent the developer from using the simulated annealing method to implement a decisional function, for example. A second solution is to use a Monte Carlo statistical analysis to determine the main trend of the random aspects of a decisional function. A third solution is to run in simulation, before run time, the decisional function, in order to determine the shape of the search space (convexity, valleys, etc).

Regarding verifiability, structural test coverage is still applicable, but does not show the complexity of the search space: a decisional function can be encoded in a few pages of source code in a specific programming language, which does not highlight the complexity of the underlying search space. Another difficulty comes from the huge size of this search space: it is impossible to check all outputs given all inputs (see section 2). A first solution resides in determining semantic zones/phases in input data. They represent sets of input values for which a meaning can be identified. This in turn determines portions of the search space, for which one solution must at least be looked for: one solution per phase. Tests can even be decomposed into categories: toy tests, medium tests, difficult tests, tests to the limit. The result of these tests may be compared with the solution provided by a Human expert in the problem encoded by the decisional function. If the Human expert is not capable of providing a solution, an assessment of the solution provided by the decisional function may be performed. A second solution is to detect unusual cases: tests are run on these cases and their quality is assessed by a Human expert, as before. A third solution slightly varies from this one: it consists in randomly applying consistent input data to a decisional function, and once again making the Human expert assess the output. A fourth solution consists in running a decisional function on a scenario --- with assessment by a Human expert.

Regarding response time (due to its exponential complexity, a decisional function may take an arbitrary large amount of time to deliver its solution), a first solution is to use anytime algorithms. These algorithms provide an output with an increasing quality as a function of time. As a result, the more time the algorithm is given, the better the quality of its output --- as opposed to no solution at all found by the decisional function. For example, decisional functions using iterative algorithms can be forced to adopt an anytime behavior --- by keeping the last solution in memory, although partial.

A second solution to response time is to adopt reactive behaviors: to each decisional function of exponential complexity, a polynomial function is associated in a lower level of the software architecture. Since this function has a polynomial complexity, it runs much faster than the decisional function it backs up. The mechanism is then to activate this backup function while the decisional function is looking for a solution in exponential time. An example of this is constituted by evasive maneuvers performed by autonomous aircraft, when the aircraft is locked by a track radar (a missile is about to be launched): they perform escaping trajectories to attempt at avoiding the detection range of a radar, while the decisional functions keeps running to produce a new action to perform.
as a justified plan for the aircraft. Of course, the main question becomes: how long will the backup function last? And especially, will it last long enough for the decisional function to return its decision on the course of actions of the aircraft? No theory exists to prove analytically this point, but experiments have been performed in simulation and show its efficiency [11].

Regarding memory space (which can grow in an arbitrarily large way, since decisional algorithms browse a tree-shaped search space), a solution is to limit memory consumption by limiting the depth of the search space that the decisional function performs. In other words, the decisional function becomes blind beyond some depth.

Let \( p \) be the depth in the tree-shaped search space, \( f \) be a function providing the number of nodes visited in the search tree as a function of the current depth (usually, it can be inversed), \( M \) be the RAM memory of the embedded computer, \( S \) be the swap memory available on the disk of that computer, and \( E \) be the memory needed to store one state in the tree-shaped search space. Then the maximal depth is given by:

\[
p = f^{-1}\left(\frac{M + S}{E}\right)
\]

Beyond this threshold, the decisional function is sure to exhaust the memory of the embedded computer. This limit can be seen as a horizon of the decisional function.

Regarding programming languages and coding, the standard Def Stan 00-55 imposes strong constraints: (i) a strongly typed language discards C; (ii) a language by block eliminates FORTRAN and its “go to” statements; (iii) a procedural language eliminates PROLOG, despite the easiness with which a decisional function can be encoded in it; (iv) the aspects of modularity, encapsulation and abstraction pushes forward languages such as Ada and C++. According to this standard, the features to avoid in a programming language are: (i) concurrency, which is entailed by the backup functions defined above; but due to the limited number of such concurrent processes, we favor the use of the above solution; (ii) interruption, which prevents from using call backs for example, and which has no use for decisional functions; (iii) recursion, which can be replaced by “while” statements and a stack for storing the intermediate states; (iv) floating point variables, which can be replaced by integer values for cost function in decisional functions for example.

6. Conclusion

In this paper, we have studied the relation between norms/standards and decisional functions. After a survey of norms and standards applicable to autonomous aircraft (aerial: DO-178B/ED-12B, SAE ARP 4754 & 4761; spatial: ECSS-E-40 Part 1B; software: MIL-STD-498, EIA-12207, DEF-STAN-00-55; safety: MIL-STD-882, DEF-STAN-00-56), we determined the common requirements of these norms and standards. They consist in analyzing the danger, risks and accident that can occur during the flight of the aircraft, and taking measures to lower the criticality to an acceptable level. We then have pointed out the characteristics of decisional functions, as needed for autonomous aircraft: non determinism, memory space, tests, and verifiability.

The critical points regarding this type of software, as opposed to classical ones, were approached using regular software life cycle, but with technical considerations regarding: (i) non determinism: Monte Carlo method, using simulation to determine the shape of the search space; (ii) tests: decomposing the input data into meaningful phases, resulting in a partition of the search space, on which one case of
each type (easy, medium, difficult, limit) is applied; taxonomy of tests and comparison with the results provided by a Human expert or at least assessment of the result by a Human expert if no solution is known; (iii) response time: use of anytime algorithm; use of backup function for ensuring survival of the aircraft while the decisional function is running (pushing critical functions to lower levels of the software architecture); (iv) memory space: limiting the depth of the tree-shaped search space and (v) rules for choosing a programming language and some features in it.

References