Tolerance of Design Faults

David Powell[1,2], Jean Arlat[1,2], Yves Deswarte[1,2], and Karama Kanoun[1,2]

[1] CNRS ; LAAS ; 7 avenue du Colonel Roche, F-31077 Toulouse Cedex 4, France
[2] Université de Toulouse ; UPS, INSA, INP, ISAE ; UT1, UTM, LAAS ; F-31077 Toulouse Cedex 4, France
{david.powell,jean.arlat,yves.deswarte,karama.kanoun}@laas.fr
http://www.laas.fr

Abstract. The idea that diverse or dissimilar computations could be used to detect errors can be traced back to Dynosius Lardner's analysis of Babbage's mechanical computers in the early 19th century. In the modern era of electronic computers, diverse redundancy techniques were pioneered in the 1970's by Elmendorf, Randell, Avižienis and Chen. Since then, the tolerance of design faults has been a very active research topic, which has had practical impact on real critical applications. In this paper, we present a brief history of the topic and then describe two contemporary studies on the application of diversity in the fields of robotics and security.

Keywords: design-fault, software-fault, vulnerability, fault-tolerance, recovery blocks, N-version programming, N-self-checking components.

Faults introduced into a computing system during its development[1] are a major concern in any critical application. The first defense against design faults is of course to attempt to avoid their the occurrence in the first place by applying the techniques of fault prevention (i.e., "careful design") and fault removal (i.e., "verification and testing"). Such a *fault avoidance* strategy is the basic tenet of quality management, i.e., of "getting it right the first time" [19] by building a system with zero defects. This is a laudable objective, but one whose satisfaction is difficult to guarantee with sufficient confidence, at least for non-trivial systems, when the conseqences of failure are extreme in either economic or human terms. In highly-critical applications, it is reasonable to assume that the system will indeed contain residual design faults and then apply *fault tolerance* (i.e., "diverse redundancy") and *fault forecasting* (i.e., "dependability evaluation") techniques so that it can be deemed to be dependable despite faults, i.e., to complement fault avoidance by a *fault acceptance* strategy [11].

This paper is primarily concerned with the *tolerance* aspect of fault acceptance, i.e., how to introduce so-called "diverse" redundancy into a system that

[1] Development faults can be introduced during specification, design or implementation. In accordance with common usage, we generically refer to such faults as *design faults*.

C.B. Jones and J.L. Lloyd (Eds.): Festschrift Randell, LNCS 6875, pp. 428–452, 2011.

can be managed in such a way as to decrease the likelihood of system failure due to residual design faults. We first give a brief history of this concept and then describe two contemporary studies concerning:

- the tolerance of deficiencies in planning domain knowledge for robotics;
- the tolerance of security vulnerabilities in commercial off-the-shelf operating systems for aviation applications.

1 A Concise History of Design-Fault Tolerance

In this section, we first give a brief overview of the main approaches that have been studied for building systems that can tolerate residual design faults and then summarize the substantial amount of work that had been devoted to evaluating the dependability that such design-fault tolerance techniques can procure. To illustrate practical applications of the technology, we then describe two critical computer-based systems in which design-fault tolerance has been deployed.

1.1 Design-Fault Tolerance Techniques

The notion of using redundant dissimilar computations to detect errors appears to have first been proposed in the 19th century in the context of Babbage's mechanical computers [49]. In the modern era of electronic computers, the notion of design diversity for design-fault tolerance was formulated in the 1970's [26,65,9]. It has been notably used in critical systems to enhance safety (i.e., to provide a fail-safe behaviour) or availability (i.e., to ensure service continuity).

Redundant components produced in accordance with the design diversity approach are usually called *variants*. In addition to the existence of at least two variants, design-fault tolerance requires a decision maker (often referred to as the *adjudicator*) providing a supposedly error-free result from the execution of the variants. The specification common to all variants must explicitly mention the *decision points*, i.e., a) when the decisions must be taken, and b) the data on which the decisions must be made, hence the data handled by the adjudicator.

There exist three main approaches for design-fault tolerance using design diversity [47]: recovery blocks, N-version programming and N-self-checking software. These approaches can be seen as resulting from the application to software of three basic types of error processing [11]: backward error recovery, fault masking, and error detection and recovery. When continuity of service is not mandatory, an alternative pragmatic approach is simply to attempt to detect an erroneous task as early as possible in order to suspend it and prevent propagation of error(s); this approach is often referred to as "fail-fast" [29,14]. Error detection is provided by executable assertions relating to the data processed, the intermediate results, or the task outputs. The implementation of these assertions is based on a defensive programming style and error processing is generally implemented through exception handling. In the subsequent paragraphs, we elaborate further on the three above-mentioned diversity-based approaches.

In recovery blocks [65,5], variants are referred to as alternates and the adjudicator is an acceptance test applied to the results produced by the alternates. Thus, if the results provided by the primary alternate fail the acceptance test, the secondary alternate is executed and so on, up to the satisfaction of the acceptance test or the exhaustion of the available alternates, in which case the recovery block is considered as globally failed (still, a fail-safe behaviour is enforced).

For N-version programming [9,8], variants are referred to as versions, and the adjudicator casts a vote on the results provided by all versions. In this case, the decision is carried out cooperatively among the versions involved, instead of by means of an absolute check with respect to a pre-defined acceptance test.

N-self-checking software [47] can be viewed as a hybrid technique based on the two previous ones. In this case, at least two self-checking software components (SCSC) are executed in parallel. Each SCSC can be made up of either the association of a variant and an acceptance test, or of two variants and a comparison algorithm. In the latter case, only one of the variants (main) is required to implement the expected function, the other variant can either implement the function with reduced accuracy, or compute the reverse function (whenever possible), or even simply operate on some intermediate results of the main variant as in the so-called certification path approach [69].

Recovery blocks and N-version programming have been extensively investigated in the literature and have both prompted numerous variations or combinations, such as consensus recovery blocks [67], distributed recovery blocks [41] or $t/(n-1)$ variant programming [77]. This trend of work has been further developed via the concept of Coordinated Atomic Actions (CA-actions) that provide a unified scheme for coordinating complex concurrent activities and supporting error recovery between multiple interacting components in distributed object systems. CA-actions integrate and extend two complementary concepts: conversations and transactions (e.g., see [66,15]). Although N-self-checking software has been less extensively studied in academic circles, it is nevertheless a rather common approach in operational systems [74]. Typical examples will be provided in Section 1.3.

In the previous paragraphs, we have focused on different aspects of *software* design-fault tolerance. One can equally apply design diversity to provide protection against design faults affecting complex *hardware* components, such as processors. Accordingly, distinct processors can be used to implement the redundant channels (e.g., see [78]).

Another option for coping with design faults is *data diversity* [3]. The rationale relies on the observation that residual software faults inducing failures during operational life are usually "subtle" faults (also known as *Heisenbugs* [29]) whose activation is caused by rare situations corresponding to certain values in the input space (at large) — e.g., operation in a specific range of values, operating system induced memory leaks, etc.) — that are difficult to reproduce. The idea is that such failures could be averted by applying a minor "perturbation" to the input data, that would still be acceptable for the software. A very "extreme" approach is known as *environment diversity* [37], which encompasses several progressive procedures:

retry, restart and reboot. A specific form of environment diversity has been popularized under the label *software rejuvenation* [34,18]. Thus, data diversity works in cases when a single channel (thus relying on time redundancy) or identically-redundant (i.e., non dissimilar) channels are executing on distinct data. It is thus cheaper to implement than design diversity techniques. Data diversity has also been intensively used for coping with the risk of common mode failures resulting from faults affecting hardware components in redundant channels (e.g., see [62]). Recently, data diversity has also been proposed as a suitable approach for coping with malicious intrusions (e.g., see [60]).

1.2 Design-Fault Tolerance Evaluation

The main objective when using design-fault tolerance approaches is to enhance system dependability attributes (such as safety or availability). However, the development and operation of design-fault tolerant systems: i) usually induce a significant cost overhead, essentially in terms of development and maintenance effort, and ii) for some techniques, give rise to a non-negligible increase in the execution time (i.e., a degradation in performance). It is therefore very important to determine what types of benefits can be expected from using design-fault tolerant systems, and to assess the positive as well as the negative impacts of design-fault tolerance approaches. In this section, we give a global overview of the kind of assessments that have been performed to characterize design-fault tolerance approaches as well as the varieties of results that can be obtained from these assessments. Emphasis is mainly put on their impact on system dependability. This overview is not exhaustive due to the huge amount of literature related to the topic, dating back to the early 1980's .

Assessment is carried out based on: i) statistical modeling approaches, or ii) controlled experiments, or iii) field data related to real-life systems. The latter two approaches are usually referred to as empirical studies. Each kind of approach has its advantages and limitations. Obviously, results based on field data are more significant than those based on experimentation and modeling. Unfortunately, there are few such analyses, because they require the collection of very specific data, most of the time related to (hopefully) very rare events, hence requiring a long period of data collection to lead to meaningful results.

It is well known that the primary causes of N-version programming failures are due to correlated faults in versions and precision problems in the voting components. For recovery-block programming, the main causes of failures correspond to related faults between the variants or between one variant and the acceptance test (see, e. g., [7] for more details). For self-checking software components, the main cause of failure is due to the inefficiency of the self-checking algorithms.

Not surprisingly, as far as we are aware, all published studies, be they based on modeling or on experimentation, show that the effectiveness of design-fault tolerance techniques is strongly related to: i) the degree of independence of the versions and the effectiveness of the decision or voting mechanism, in the N-version programming technique, ii) the effectiveness of the acceptance test for the recovery block techniques, and iii) the effectiveness of the error detection

and recovery mechanisms in N-self-checking software. However, even though the above general results are agreed on, in some papers they are presented either i) in an optimistic way to show the superiority of design-fault tolerance techniques compared to fault-avoidance techniques, or ii) in a pessimistic way to show their limitations. For example, even if an early experimental work [43] was considered as controversial (and is usually referenced as such), the experimental results confirm the above general conclusions. Moreover, based on the results published in the same paper, but processing the data differently, a subsequent study [31] showed that there is a substantial reliability improvement for the N-version programming system compared to a single software system.

Consequently, the difficulty in quantifying the effectiveness of design-fault tolerance techniques lies on the characterization of the above limiting factors and of their impact on the global system dependability. Modeling has been carried out based on various techniques, such as state graphs or Markov chains [30,46,48], fault trees [23], stochastic reward nets [71], software execution schema (to identify the combination of fault activations leading to failure) [7,70], Bayesian approaches [51], the derivation of the intensity of coincident errors [25,61], or the derivation of score functions [64]. These evaluations either compare various design-fault tolerance techniques and give conditions under which one technique is superior to other(s) or to a single version, or concentrate on a single design-fault tolerance technique to provide conditions under which this technique is a better strategy than a single version.

Modeling approaches require several assumptions to build tractable models that can be processed to provide usable results. Some assumptions are more realistic than others. Additionally, modeling requires numerical values of some parameters that are not all available from field data. More assumptions are thus needed related to the numerical values of the parameters. Nevertheless, sensitivity analyses with respect to the assumptions as well as to numerical values of the parameters lead to interesting results. Moreover, they allow the identification of the most significant parameters, which should be evaluated accurately to correctly quantify the practical dependability gain induced by design-fault tolerance.

Some controlled experiments have been carried out on software programs developed by professional programmers while, in other experiments, the programs have been developed in academic environments. As far as we are aware, experiments involving professional programmers are rare. They are based on a small number of versions. For example: i) in [4], only a single software version was developed and the experiments compare the reliability obtained with that single version when fault-tolerance is enabled with that obtained when fault-tolerance is disabled, and ii) in [16] three versions have been developed to analyze the impact of the N-version programming approach to software-fault tolerance, including back-to-back testing.

Experiments carried out in academic environments have, in general, been applied to relatively small investigational software systems, developed by less experimented programmers (most of the time, by students) because of cost

considerations, thus under conditions that are sometimes far from industrial development conditions. Usually these experiments involved a large number of software programs, allowing several analyses to be carried out, based on the combination of several versions, back-to-back testing, fault injection, etc.

Significant sets of experiments have been carried out since the 1980's (see, e.g., [12,43,39,50,24]) and continued until more recently [76,56]. A major advantage of controlled experiments concerns the validation of modeling approaches [67,75] as well as some of the assumptions made by the various conceptual models.

Finally, some analyses addressed cost overhead induced by design-fault tolerance, either based on modeling [57] or on experimentation (see, e.g., [4,38]). For example, based on observations of a real-life fault tolerant software system, the latter reference showed that the global cost overhead from functional specifications to system test ranges between 42% to 71%, excluding the requirement specifications phase that is performed only once for the whole software system. This overhead has been estimated to be 60% in [4].

1.3 Design-Fault Tolerance Applications

Modern passenger aircraft, such as the Airbus 320/330/340 family [17] and the Boeing 777 [78], include computers in the main flight control loop to improve overall aircraft safety (through stability augmentation, flight envelope monitoring, windshear protection, etc.) and to reduce pilot fatigue. Of course, these increases in aircraft safety must not be annihilated by new risks introduced by the computing technology itself. For this reason, the flight control systems of these aircraft are designed to be fault-tolerant and accordingly they rely extensively on design diversity. Railway applications are also increasingly relying on computer technology, in particular speed regulation and rail-traffic control. It is clear, here also, that the stringent safety requirements have resulted in the development of fault-tolerant architectures, incorporating hardware redundancy and often also software redundancy (e.g., see [32,6]). The subsequent paragraphs describe two typical examples in these application domains.

The Airbus 320 Flight Control System. Fault tolerance in the flight control system of the Airbus 320/330/340 family is based on the error detection and compensation technique [11], using the N-self-checking software approach described in Section 1.1. Each flight control computer is designed to be self-checking, with respect to both physical and design faults, to form a fail-safe subsystem.

Each computer consists of two lanes supporting functionally-equivalent, but diversely-designed programs (Figure 1).

Both lanes receive the same inputs, compute the corresponding outputs and check that the other lane agrees. Only the control lane drives the physical outputs. Any divergence in the results of each lane causes the physical output to be isolated. Each flight control axis of the aircraft can be controlled from several such self-checking computers.

The complete set of computers for each axis processes sensor data and executes the control loop functions. However, at any given instant, only one computer in

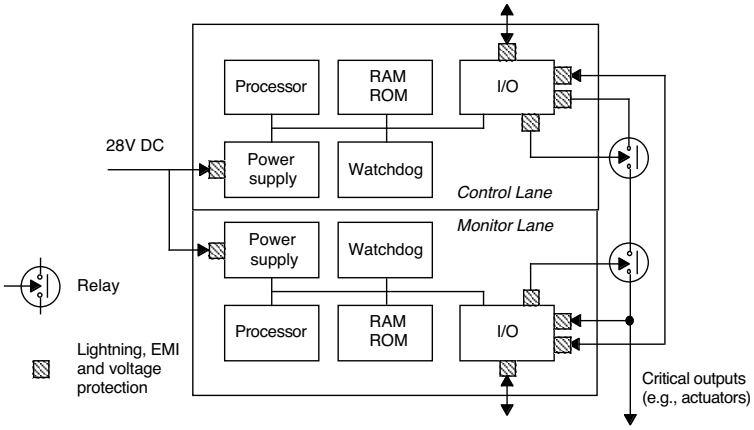

Fig. 1. Self-checking computer featuring diversely-designed control and monitor lanes (adapted from [72])

the set (the primary) is in charge of physically controlling the actuators. This computer sends periodic"I'm alive" messages to the other computers in the set, so that they may detect when it fails. Should the primary fail, it will do so in a safe way (thanks to the built-in self-checking) without sending erroneous orders to the actuators. According to a predetermined order, one of the other computers in the set then becomes the new primary and can immediately close the control loop without any noticeable jerk on the controlled surface.

The design diversity principle is also applied at the system level. The set of computers controlling the pitch axis (Figure 2) is composed of four self-checking computers: two Elevator and Aileron Computers (ELACs) and two Spoiler and Elevator Computers (SECs), which are based on different processors and built by different manufacturers. Given that each computer type supports two different programs, there are overall four different pitch control programs.

There is also considerable functional redundancy between the flight control surfaces themselves. Accordingly, it is possible to survive a complete loss of all computer control of some surfaces, as long as the failed computers fail safely. Furthermore, if all computers should fail, there is still a (limited) manual backup. Similar system-level diversity has been implemented in subsequent planes of the Airbus family; it also encompasses the power supply chains and the hydraulic systems, see [72].

The ASF Computer Based Interlocking System. We consider the case of the system developed by *Ansaldo Segnalamento Ferroviario* for the control of rail traffic [58] and that manages the Termini station in Rome. As most systems of this type, the *Computer Based Interlocking* (CBI) system implements a hierarchical architecture (Figure 3):

– *level 1*: a control centre equipped with *workstations* (*WS*) for managing the station and supervising the diagnosis and maintenance operations;

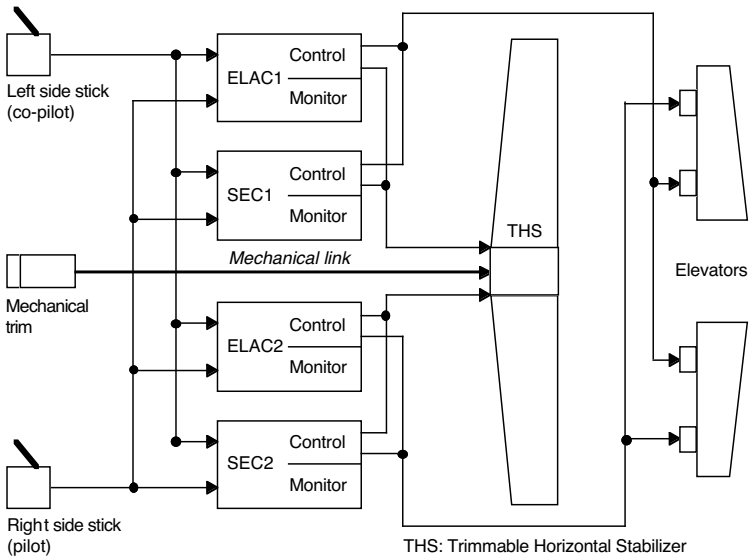

Fig. 2. Sketch of the Airbus 320 pitch control (adapted from [72])

- *level 2*: a central processing unit, or *safety nucleus* (SN) of the CBI, in charge of the overall management of the safety operations for the station concerned;
- *level 3*: a set of remote computers, called *trackside units* (TU), intended for data acquisition and application of the orders provided by the SN processors (localisation of the trains, operation of signals and switches).

Various forms of redundancy are used for each level:

- WS: simple dynamic redundancy,
- SN: triple modular redundancy (TMR) featuring code and data diversification for each of the replicated sections $S_i, i = 1..3$
- TU: self-checking redundancy (pair of computers).

We focus hereafter on the SN level, which constitutes the backbone of the CBI. The three redundant sections of the SN elaborate a logical decision based on standard majority voting and rely on a self-checking hardware device (*exclusion logic*) that ensures the exclusion of the TMR module (section) declared as failed. Each section is isolated from the others and has its own power supply. It consists of a logical unit (LU) and a peripheral unit (PU). The SN layer thus consists of six processing units interconnected by means of dedicated serial lines, with optical coupling. The software architecture of each module is similar, with the operating system, the applications and the data stored in read-only memory. However, the application software is diversified: distinct programmer teams, dissimilar languages, different coding of the data, storage of the code and

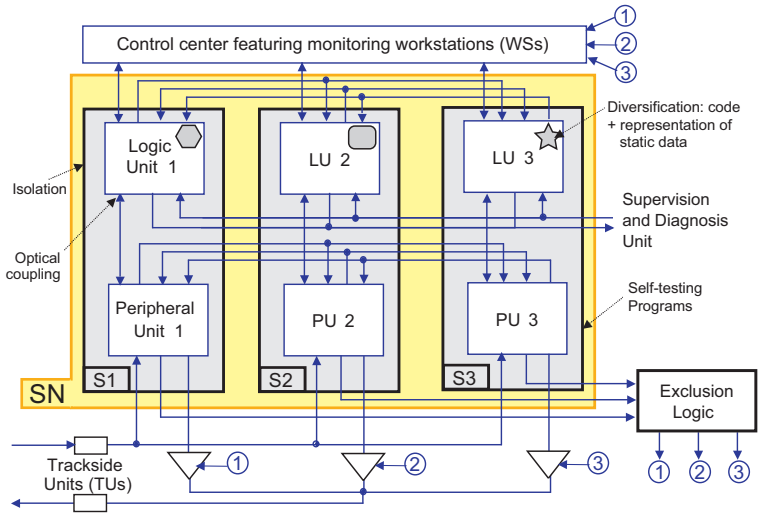

Fig. 3. The CBI Safety Nucleus (adapted from [58])

data segments at different address locations in each section. In addition, self-testing routines are executed in each section to provide a form of "watchdog" monitoring.

The *SN* layer is designed to provide both availability (via its TMR structure) and safety (based on a "2-out-of-2 vote" to be described later). The elimination of the first failed section by the exclusion logic scheme removes the risk of

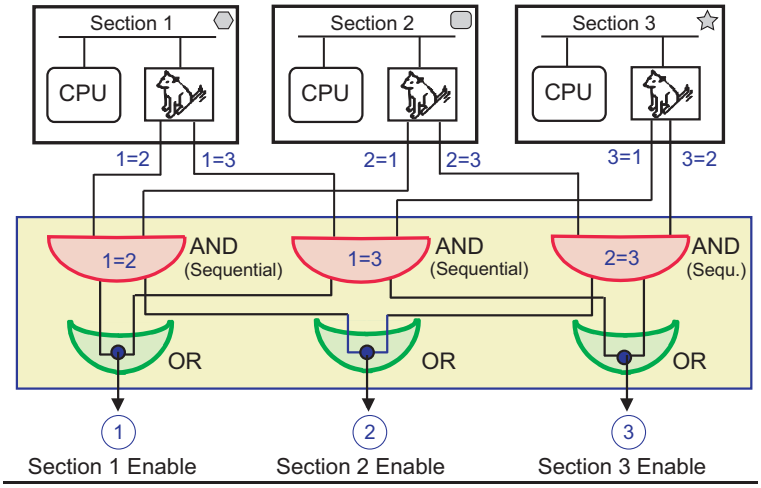

Fig. 4. Principle of the exclusion logic (adapted from [58])

common mode failure that could result from the subsequent failure of one of the two remaining operational sections. Furthermore, the isolated section can be readily accessed for diagnosis and maintenance, and later reintegration into the operational system.

The principle of the exclusion logic (*EL*) device is depicted on Figure 4. It consists of three identical modules that process the analog commands produced by the sections of the *SN*, after a software-implemented vote. Information concerning the agreement (resp. disagreement) of a section with the two others – active-level signal (resp. passive-level signal) – is communicated to the *EL* at the same time as the commands. Each module of the *EL* checks (sequential AND function) for an agreement between the two corresponding input signals and activates the enabling signal powering the drivers of the transmission modules of the *SN* (communication with the *WS*'s of the control centre and the *TU*'s). In case of disagreement, the power of the transmission device associated with the failed section is cut off, which isolates it from the other two sections and from the *TU*'s.

2 Tolerance of Design Faults in Declarative Models for Temporal Planning

In this section, we present a recent study on an application of design-fault tolerance in robotics [53,54]. We consider how diversification can be applied to the declarative domain models that allow robots to autonomously establish a plan of activities to reach specified goals in a partially-unknown environment.

2.1 The Problem: Deficiencies in Planning Domain Models

Fully autonomous systems, such as space rovers, robotic assistants for care of the elderly, museum tour guides, and autonomous vehicles, should be able to choose and execute high-level actions without any human supervision, in practice using planners as a central decisional mechanism. However, one of the major stumbling blocks to the adoption of these techniques in critical applications is the difficulty of predicting and validating the behavior of such decisional software, especially when faced with the open, unstructured and dynamic environments that are their *raison d'être*.

Planning is the activity of producing a plan to reach a goal (or set of goals) from a given state, using given action models. A planner typically consists of two parts: a) a declarative domain model describing domain objects, possible actions on these objects and the associated constraints, and b) a planning engine, that can reason on the domain model and produce a plan of actions enabling planning goals to be reached. Here, we consider planning carried out by searching in plan space, as carried out by the IxTeT planner [27]. In this approach, CSP (Constraint Satisfaction Problem) solving techniques are used to determine a possible evolution of the system state that satisfies a set of constraints, some of which specify the system goals. This is done by iteratively assigning possible values to each variable and verifying that all constraints remain satisfied.

At the time at which a plan is established, not everything may be known about the environment nor its future evolution. Thus, planners seek to produce *flexible* plans where as much latitude and scope for adaptation as possible are left in the plan [27,59]. Moreover, adverse situations that appear during plan execution, causing some of its actions to fail, can be tolerated through:

- *Re-planning*, which consists in developing a new plan from the current system state and still unresolved goals. Depending on the complexity of the planning model, re-planning may require a significant amount of processing. Other system activities are thus generally halted during re-planning.
- *Plan repair*, which attempts to reduce the time lost in re-planning by salvaging parts of the previous failed plan, and executing them while the rest of the plan is being repaired. However, if reducing the salvaged plan conflicts with unresolved goals, plan repair is stopped and re-planning is initiated.

In this respect, planning and plan execution can be thought of as a generalized form of forward recovery, which will attain the assigned goals if a solution plan exists and if it can be be found in time for it to be executed.

Among the various reasons that can prevent a planner from finding an adequate plan are deficiencies in the heuristics that guide the search process and deficiencies in the domain model itself, since the latter effectively defines what plans the planner can produce. Indeed, the problem of planner validation has received much attention, but it remains notoriously hard [28,33,40,63]. In the next sub-section, we describe a complementary solution that aims to *tolerate* residual planner design faults, focussing particularly on the domain model.

2.2 A Solution: Fault-Tolerant Planning

The general principle of the proposed mechanisms is to execute, sequentially or concurrently, diversified variants of the planner, following approaches similar to recovery blocks [65] and distributed recovery blocks [41]. In particular, diversity is encouraged by forcing the use of different algorithms and variable domains, and different parameters in the models and heuristics of the variants.

Implementing error detection for decisional mechanisms in general, and planners in particular, is difficult [55]. There are often many different valid plans, which can be quite dissimilar. Therefore, error detection by comparison of redundantly-produced plans is not a viable option[2]. Thus, we must implement error detection by independent means. Four complementary error detection mechanisms can be envisaged:

- a *watchdog timer* to detect when the search process is too slow or deadlocked;
- a *plan analyzer* as an acceptance test to check, before execution, that the produced plan satisfies a number of constraints and properties;

[2] For the same reason, N-version programming was excluded as a possible design-fault tolerance option.

- a *plan failure detector*, a classic plan execution control mechanism needed to detect whether a plan action has failed in order to trigger plan repair and/or re-planning;
- an *on-line goal checker* that verifies whether goals are actually satisfied as the plan is executed.

Recovery from detected errors, despite possible planner design-faults, relies on the use of two (or more) planners using diversified knowledge (i.e., search heuristics and/or domain model), managed by a coordinator component (called *FT-plan*) that is sufficiently simple to be shown (or assumed) to be fault-free. Two recovery strategies are possible, differing on whether the planners are executed in sequence (like in classic recovery blocks) or in parallel (like in distributed recovery blocks). For space reasons, we only consider here the sequential strategy (see Algorithm 1).

Basically, each time an error is detected, we switch to another planner until all goals have been reached or until all planners fail in a row from the same initial system state. In the latter case, no models allow the planner to tackle the planning problem successfully: an exception must be raised to inform the operator of mission failure and to allow the system to be put into a safe state (line 29). When all the planners have been used but some goals are still unsatisfied, we revert to the initial set of planners (while block: lines 4 to 32).This algorithm illustrates the use of all four error detection mechanisms: watchdog timer (lines 9 and 25), plan analyzer (line 14), plan failure detector (lines 16 and 18), on-line goal checker (lines 4, 6 and 17).

Until all goals have been achieved, the proposed algorithm reuses planners that may have been previously detected as failed (line 5). This makes sense since (a) a perfectly correct plan can fail during execution due to an adverse environmental situation, and (b) some planners, even faulty, can still be efficient in situations that do not cause fault activation.

2.3 Assessment

We implemented a prototype version of the *FTplan* coordinator component and integrated it in the LAAS architecture for autonomous systems [2], which has been successfully applied to several mobile robots involved in real applications. The *FTplan* prototype can coordinate a fault-tolerant configuration of two IxTeT temporal planners [27]. It implements the sequential planning strategy of Algorithm 1 and includes three of the four error detection mechanisms: watchdog timer, plan failure detector, and on-line goal checker, but not a plan analyzer (the design of which is a research topic in its own right). To evaluate the proposed approach, we used two diverse declarative models (which we call $Model_1$ and $Model_2$) for autonomous planning of the missions of a planetary exploration rover. The missions involve three sets of goals: (a) take **photos** of a set of science targets (e.g., rocks) located at pre-defined coordinates with respect to the rover's initial position; (2) **communicate** with an overhead orbiter during predefined communication windows, and (3) **return** to the initial position. While

Algorithm 1. Sequential Planning Strategy

```
1: begin mission
2:    exec_failure ← NULL
3:    failed_planners ← ∅
4:    while attainable_goals ≠ ∅ do
5:       candidates ← planners
6:       while candidates ≠ ∅ ∧ attainable_goals ≠ ∅ do
7:          choose new planner k such that (k ∈ candidates) ∧ (k ∉ failed_planners) ∧ [(k ≠
            exec_failure) ∨ (k ∪ failed_planners = candidates)]
8:          candidates ← candidates\k
9:          init_watchdog(max_duration)
10:         send (plan_request) to k

11:         wait {for either of these two events}
12:         □ receive (plan_found) from k
13:         stop watchdog
14:         if analyze(plan) = OK then
15:            failed_planners ← ∅
16:            res_exec ← k.execute_plan()
17:            update(attainable_goals)
18:            if res_exec ≠ OK then
19:               exec_failure ← k
20:            end if{if the plan fails, then attainable_goals ≠ ∅ and the on-line goal checker will
               loop to line 3 or line 4}
21:         else
22:            log(k.invalid_plan)
23:            failed_planners ← failed_planners ∪ k
24:         end if
25:         □ timeout watchdog
26:         failed_planners ← failed_planners ∪ k
27:         end wait

28:         if failed_planners = planners then
29:            raise exception 'no remaining planners' {the mission has failed}
30:         end if
31:      end while
32:   end while
33: end mission
```

navigating to the science targets and back to its initial position, the robot must move round any obstacles that it meets on this path. At the plan execution level, the unknown obstacles create uncertainty as regards the outcome of action executions, and can possibly prevent the rover from achieving some of its goals.

In our experiments, $Model_1$ was a mature model that had been successfully used several times both indoors and outdoors on a real robot. $Model_2$ was a new model, in which diversification with respect to $Model_1$ was forced through specific design choices. For example, the rover's position is defined in $Model_1$ using Cartesian coordinates whereas $Model_2$ uses a symbolic representation, thus implementing fundamentally different algorithms than those of the former. Both models contain action descriptions for:

- movement initialization (allowing, in particular, recovery from any earlier navigational errors),
- camera initialization,
- moving the camera,
- taking a photo,

– communicating with the orbiter,
– moving the rover from point A to point B.

In $Model_1$, the points A and B may be anywhere, whereas in $Model_2$ they are specific symbolic points (such as the initial position of the system and the positions of different goals). A second move action is defined with $Model_2$ for the case where the rover needs to move from an arbitrary point to a specific one (typically needed when a previous move has failed).

To evaluate the efficacy of the fault-tolerance approach, we compared the behavior of two simulated rovers in the presence of injected faults:

– *Robot1*: using $Model_1$ in a non-redundant planner
– *RobotFT*: using both $Model_1$ and $Model_2$ in redundant planners coordinated by *FTplan*.

Faults were injected into $Model_1$ on both robots by means of the Sesame mutation tool [20] . The rovers were faced with four increasingly complex *worlds* (set of obstacles): $W1 - W4$ and given four increasingly complex *missions* (sets of photos and communication windows): $M1 - M4$, leading to a total of 16 world $\times$ mission combinations. Figure 5 illustrates the most complex combination: W4M4.

The experiments are inherently non-deterministic, due to asynchrony of the various robot subsystems and in the underlying operating systems. Task scheduling differences between similar experiments may degrade into task failures and

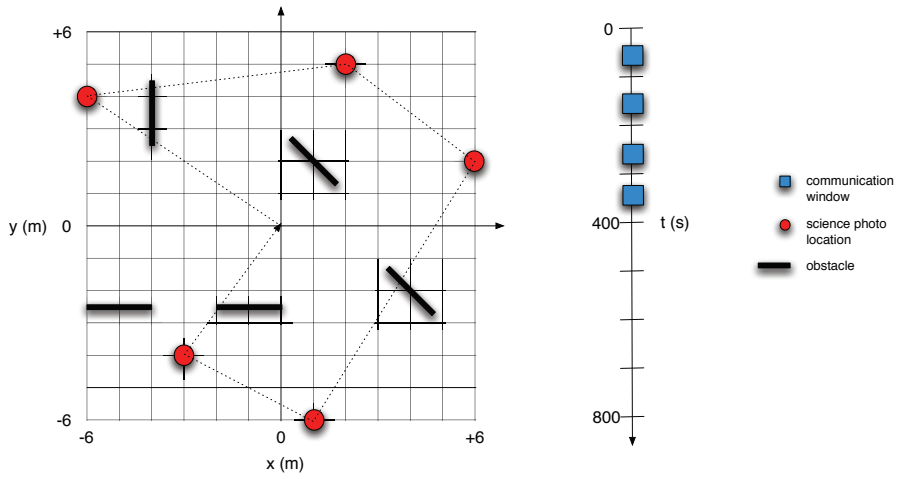

Fig. 5. World W4M4. The mission starts and ends in the center of a $12m \times 12m$ grid. The robot should visit the 5 science photo locations, send data to the orbiter during the 4 communication windows and avoid the 5 long thin obsacles. The dotted line indicates a possible robot itinerary *ignoring* obstacles.

possibly unsatisfied goals, even in the absence of faults. To address this non-determinacy, each basic experiment was executed three times, leading to a total of 48 experiments per fault scenario. More repetition would of course be needed for statistical inference on the basic experiments, but this would have led to a total number of experiments higher than that which could have been carried out with our available resources (including initialization and data treatment, each basic experiment lasts about 20 minutes).

The overall results for 28 different mutations are summarized in Figure 6 (for detailed results, see [53,54]). The figure shows bar graphs of the percentages of satisfied goals and missions averaged over all four missions. We give separate averages for the three simpler worlds ($W1-W3$) and for all four worlds together ($W1-W4$), since it was often the case that even a fault-free planner was unable to find a solution plan for world $W4$, with its numerous long thin obstacles (cf. Figure 5).

These results show that, with the considered faultload:

- The redundant diversified models of the fault-tolerant *RobotFT* provide an notable improvement to dependability in the presence of faults (bar graphs labelled F): in all cases, the proportions of failed goals decrease compared to the non-redundant *Robot1*.
- Even when considering the pessimistic measure of the proportion of failed missions (a mission is considered as failed even if only a single elementary goal is not achieved), the improvement procured by redundant diversified models is appreciable: 41% in worlds $W1-W3$, 29% when world $W4$ is also considered.

Note, however, that in the presence of injected faults, the fault-tolerant *RobotFT* is *less* successful than a single fault-free model (compare *RobotFT* of the bar graphs labelled F, with *Robot1* of the bar graphs labelled ∅). This apparent

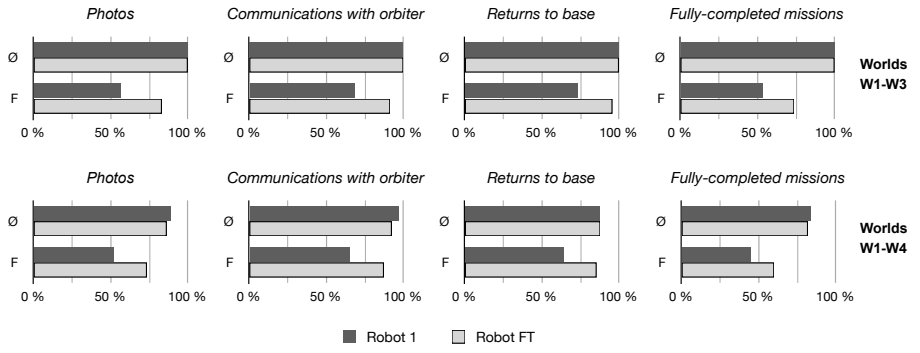

Fig. 6. Effectiveness of fault-tolerant planner (*RobotFT*) compared to baseline planner (*Robot1*). *The bar graphs show the percentages of satisfied goals (photos, communications and returns) and completed missions, without* (∅) *and with* (F) *injected faults.*

decrease in dependability can be explained by the fact that incorrect plans are only detected when their execution has failed, possibly rendering one or more goals unachievable, despite recovery. This underlines the potential improvement that could be obtained if *FTplan* were to include a plan analyzer to detect errors in plans *before* they are executed.

3 Tolerance of Security Vulnerabilities in COTS Operating Systems

This section presents an ongoing study on the use of design-fault tolerance techniques to ensure security. Specifically, we consider how diversification can be applied to provide protection against security vulnerabilities in commercial operating systems when used in a critical application for the aviation industry.

3.1 The Problem: Operating System Vulnerabilities

Nowadays, the main threat against computer security is represented by malicious software (malware) exploiting design flaws of existing commercial off-the-shelf (COTS) operating systems. Indeed, COTS operating systems are too complex to be free from design faults, despite some attempts to apply formal development methods to at least parts of them (e.g., seL4 [42]). And even if formal verification methods are applied, it is difficult to prove the absence of vulnerabilities: there is no model suitable for expressing both the desired high-level properties and the implementation of subtle hardware mechanisms (address space control [45], interrupt-handling, or hardware management functions such as ACPI (Advanced Configuration and Power Interface) [22], or even protection mechanisms [52]). For instance, formal approaches that apply successive refinements of models (e.g., the B method [1]) cannot descend to the level of COTS processors [36].

Nevertheless, COTS operating systems are attractive, even for moderately critical applications, because they offer advanced facilities at a low cost, enabling the easy development of efficient and high performance applications. They are used in many different environments by very large numbers of people, and their reliability increases with their maturity (design flaws are frequently identified and corrected). On the other hand, the wide distribution of COTS operating systems make it easy for potential attackers to obtain detailed knowledge of their functionalities and vulnerabilities.

As for other design faults, tolerating COTS OS vulnerabilities relies on execution diversity and adjudication of execution results to provide an error-free result or to stop the execution (to provide a fail-safe behaviour). One way to implement OS diversity is to develop diverse variants of applications, executed on different operating systems, running on different hardware platforms [21]. But such an implementation would be more complex than necessary: to tolerate OS vulnerabilities, the same application can be executed on different operating systems. For that, the application needs to be developed independently of the operating system that will execute it, by interfacing a generic library API, or a

generic execution environment. This is the case for instance of Java programs, designed to be run in a Java Virtual Machine (JVM), each operating system implementing its own version of the JVM.

Similarly, it is not necessary to run the different operating systems on different hardware platforms: most hardware platforms (Intel, AMD, ARM, etc.) support several different operating systems. Moreover, in some cases it would be very inconvenient for a user to cope with several independent and redundant human-machine interfaces (screens, keyboards, mice, etc.) while interacting with a single application. A better solution is thus to run a single application software over diverse operating systems run by a single hardware platform. This is possible, thanks to virtualization techniques, as explained in the next subsection.

3.2 A Solution: OS Diversification through Virtualization

Virtualization techniques were introduced for the first time by IBM in the seventies [73] to emulate, for a given software component, the operation of the underlying hardware. Over the last decade, these techniques have been studied and developed to offer standardized layers of virtualization. For instance, in a multitask operating system, every task has a "virtualized view" of the hardware layer, since every task considers that it is running alone on the hardware. For operating system diversification, we are interested in the notion of "system virtual machine" [68], which allows many operating systems to be used in parallel on the same physical machine (cf. Figure 7).

Figure 7 represents two types of Virtual Machine Monitors (*VMM*'s). Type 2 *VMM*'s are widespread. A type 2 *VMM* is installed as an application on a host system, which can then launch and manage different operating systems. One example of such a type 2 *VMM* would be, for example, the use of VMWare on

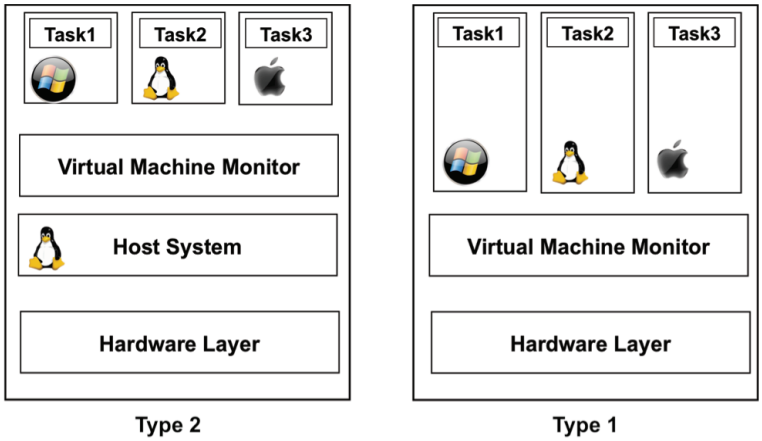

Fig. 7. Type 1 and Type 2 Virtual Machine Monitors

Linux to run a Windows operating system, but this does not help to tolerate vulnerabilities of the host operating system (here Linux). More recently, type 1 *VMM*'s (known also as hypervisors) have been developed to support different operating systems while running directly over the bare hardware. One example of such hypervisors is Xen [13].

The hypervisor technique offers complete isolation between the different virtual machines, and also a means to control the virtual machines and their interactions with input-output devices. In particular, when performing the same application task simultaneously in two or more virtual machines with different operating systems, the hypervisor can intercept all outputs from the different virtual machines, compare them and decide which copy is to be transmitted to the hardware outputs. Similarly, all inputs can be replicated by the hypervisor before sending one copy to each virtual machine.

This virtualization technique has been proposed to implement aircraft maintenance tasks on a laptop [44]. Maintenance operations present an important economic challenge for airline companies since these operations determine the time needed by ground technicians to authorize take-off of the concerned aircraft. The shorter this time, the more profitable the aircraft for the company. Aircraft maintenance is ensured by operators who, using procedures set out in maintenance manuals, analyze the aircraft flight reports and launch tests on the components detected as potentially faulty. Currently, many maintenance manuals are still not electronic, and even the electronic ones are prohibited from interacting directly with the aircraft in order to prevent any risk of corruption of critical components. The presence of a human operator is currently the only guarantee that maintenance operations follow a safe procedure that does not corrupt critical systems.

For future aircraft, it is envisaged that the manuals will be stored on a Maintenance Laptop (*ML*), which could be connected to the aircraft through Ethernet or wireless connections. The maintenance operator would use the *ML* and move freely in and around the aircraft. He would inspect directly the components identified as faulty in flight reports, follow the procedures indicated on the *ML*, and launch directly from the *ML* the set of tests that he formerly launched from specific embedded devices. For this application, the main security focus is communication between the maintenance laptop (running diversified COTS operating systems) and the aircraft. In particular, the aircraft components should be protected from illicit modification due to a compromised operating system. Similarly, the human-machine interface (*HMI*) should be protected from falsification by a compromised operating system.

An *ML* prototype has been developed, using the Xen hypervisor. The prototype consists of two identical copies of the maintenance task (written in Java), respectively Task 1 and Task 1', each running on its own virtual machine (*VM*) with a different OS, and a human-machine interface, called *MLI* (*ML* Interface) in *Domain0* of Xen. *Domain0* also supports result adjudication by a validation object (*VO*) that compares the outputs from Task 1 and Task 1' (see Figure 8). The *MLI* can be described as follows:

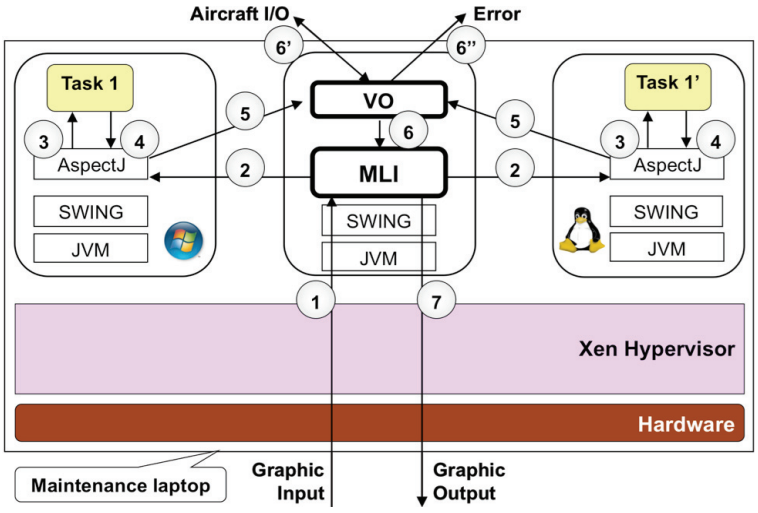

Fig. 8. Maintenance Laptop Implementation

- *MLI* provides the same GUI (Graphic User Interface) as in Task 1 and Task 1' programs. *MLI* has the same graphical components, with the same graphic charter.
- *MLI* does not implement any function of Task 1 or Task 1'. It captures, through an AspectJ wrapper, all *ML* inputs (keyboard, mouse, etc.) and forwards them to the COTS *VM*'s (cf. (1) (2) and (3) on Figure 8.

Once these inputs are forwarded to the COTS *VM*'s, Task 1 and Task 1' process the inputs. Their results, consisting of graphic outputs and data to be sent to the aircraft, are then captured (4) and forwarded to *VO* (5) for comparison. If there is a difference between the two results, this would mean that at least one *VM* is potentially corrupted, and is not generating the output it should. In this case, both COTS *VM*'s are stopped and an alert (6") is displayed to the maintenance operator (the operator would then reboot the *ML*, change his laptop, or give up using a laptop and use the traditional on-board maintenance equipment). If no difference is detected, the result is forwarded to *MLI* if it is a graphic display output (6), or to the aircraft otherwise (6'). The validated graphic output is then displayed by the *MLI* (7).

3.3 Assessment

A proof-of-concept prototype of the maintenance laptop has been developed. It aims to demonstrate the application of OS diversity within a single computer to tolerate COTS OS vulnerabilities. The idea is that when the same laptop is used for tasks other than maintenance (e.g., browsing the Internet), it can be

contaminated by malware (e.g., Trojan Horses, viruses, worms, etc.) that can compromise a COTS operating system. We claim that:

1. It is much more difficult to develop a malware able to compromise several operating systems than one able to compromise a single OS.
2. Even if a malware was able to corrupt both operating systems running the two copies Task 1 and Task 1', it would have a different effect on Task 1 and Task 1' outputs, and thus the errors would be detected.

This claim has been validated by various OS attack experiments on the *ML* prototype, e.g., changing the values to be displayed on the screen or the parameters sent to or received from the aircraft.

However, such an implementation would not tolerate a malware corruption of the hypervisor (here, Xen and its *Domain0*). We claim that it would be much more difficult to compromise such a hypervisor than an operating system:

1. A hypervisor is usually much simpler than an operating system, and thus much easier to validate. Green Hill Integrity 178B and Polixene are examples of separation kernels and hypervisors that have been evaluated respectively at EAL6 and EAL5 according to Common Criteria [35].
2. For this kind of application, the hypervisor is only active during maintenance operations: before a maintenance operation, the laptop is booted in a secure way to launch first the hypervisor and then create the virtual machines that each load its own operating system. The hypervisor itself is protected from a corrupted *VM* by its protection and separation mechanisms. When the laptop is used for non-critical operations (e.g., for Internet browsing), it is booted in the standard way, with a COTS operating system, which can be compromised but cannot access the hypervisor.

4 Conclusion

The pioneering work of Elmendorf, Randell, Avižienis and Chen in the 1970's [26,65,9], on the notion of diversity to allow tolerance of design faults, laid the foundations for over four decades of research. Despite the debate as to whether it is cost-effective and worthwhile [43,31,76], we claim that design-fault tolerance is not only beneficial, but well nigh indispensable when verification and testing cannot be carried out with a degree of confidence that is commensurate with the criticality of the considered applications. The industrial applications described in Section 1.3 of this paper, and the recent research described in Sections 2 and 3, give credence to this claim. We also claim that "info-diversity", like biodiversity, will be a necessary condition for survival of the human species in an ever-changing world in which so many aspects of our lives are now fully-dependent on information technology.

References

1. Abrial, J.: The B-Book - Assigning Programs to Meanings. Cambridge University Press, Cambridge (1996)
2. Alami, R., Chatila, R., Fleury, S., Ghallab, M., Ingrand, F.: An architecture for autonomy. International Journal of Robotic Research 17(4), 315–337 (1998)
3. Amman, P.E., Knight, J.C.: Data diversity: An approach to software fault tolerance. IEEE Trans. on Computers 37(4), 418–425 (1988)
4. Anderson, T., Barrett, P., Halliwell, D., Moulding, M.: Software fault tolerance: an evaluation. IEEE Trans. on Software Engineering SE 11(12), 1502–1510 (1985)
5. Anderson, T., Lee, P.: Fault Tolerance - Principles and Practice. Prentice-Hall, Englewood Cliffs (1981)
6. Arlat, J., Kanekawa, N., Amendola, A., Dufour, J.L., Hirao, Y., Profeta III, J.: Dependability of railway control systems. In: 16th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-16), pp. 150–155. IEEE CS Press, Vienna (1996)
7. Arlat, J., Kanoun, K., Laprie, J.C.: Dependability modeling and evaluation of software fault-tolerant systems. IEEE Trans. on Computers 39(4), 504–513 (1990)
8. Avižienis, A.: The N-version approach to fault-tolerant systems. IEEE Trans. on Software Engineering 11(12), 1491–1501 (1985)
9. Avižienis, A., Chen, L.: On the implementation of N-version programming for software fault tolerance during execution. In: 1st IEEE-CS Int. Computer Software and Applications Conference (COMPSAC 1977), pp. 149–155. IEEE CS Press, Chicago (1977)
10. Avižienis, A., Kelly, J.: Fault-tolerance by design diversity: Concepts and experiments. Computer 17(8), 67–80 (1984)
11. Avižienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic concepts and terminology of dependable and secure computing. IEEE Trans. on Dependable and Secure Computing 1(1), 11–33 (2004)
12. Avižienis, A., Lyu, M., Schutz, W., Tso, K., Voges, U.: DEDIX 87 - a supervisory system for design diversity experiments at UCLA. In: Voges, U. (ed.) Software Diversity in Computerized Control Systems, pp. 129–168. Springer, Wien (1988)
13. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. In: 19th ACM Symp. on Operating Systems Principles (SOSP), pp. 164–177. ACM, New York (2003)
14. Bartlett, J., Gray, J., Horst, B.: Fault tolerance in Tandem computer systems. In: Avižienis, A., Kopetz, H., Laprie, J.C. (eds.) The Evolution of Fault-Tolerant Systems, pp. 55–76. Springer, Vienna (1987)
15. Beder, D., Randell, B., Romanovsky, A., Rubira, C.: On applying atomic actions and dependable software architectures for developing complex systems. In: 4th IEEE Int. Symp. on Object-Oriented Real-Time Distributed Computing, pp. 103–112. IEEE CS Press, Magdeburg (2001)
16. Bishop, P., Esp, D., Barnes, M., Humphreys, P., Dahl, G., Lahti, J., Yoshimura, S.: Project on diverse software - an experiment in software reliability. In: Safety of Computer Control Systems (SAFECOMP), pp. 153–158 (1985)
17. Brière, D., Traverse, P.: Airbus A320/A330/A340 electrical flight controls - a family of fault-tolerant systems. In: 23rd IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-23), pp. 616–623. IEEE CS Press, Toulouse (1993)
18. Castelli, V., Harper, R., Heidelberger, P., Hunter, S., Trivedi, K., Vaidyanathan, K., Zeggert, W.: Proactive management of software aging. IBM Journal of Research and Development 45(2), 311–332 (2001)

19. Crosby, P.B.: Cutting the cost of quality; the defect prevention workbook for managers. Industrial Education Institute, Boston (1967)
20. Crouzet, Y., Waeselynck, H., Lussier, B., Powell, D.: The SESAME experience: from assembly languages to declarative models. In: Mutation 2006 - The Second Workshop on Mutation Analysis, 17th IEEE Int. Symp. on Software Reliability Engineering (ISSRE 2006). IEEE, Raleigh (2006)
21. Deswarte, Y., Kanoun, K., Laprie, J.C.: Diversity against accidental and deliberate faults. In: Amman, P., Barnes, B., Jajodia, S., Sibley, E. (eds.) Computer Security, Dependability and Assurance: From Needs to Solutions, pp. 171–182. IEEE CS Press, Los Alamitos (1999)
22. Duflot, L., Levillain, O., Morin, B.: ACPI: Design principles and concerns. In: Chen, L., Mitchell, C., Martin, A. (eds.) Trust 2009. LNCS, vol. 5471, pp. 14–28. Springer, Heidelberg (2009)
23. Dugan, J., Lyu, M.: Dependability modeling for fault-tolerant software and systems. In: Lyu, M. (ed.) Software Fault Tolerance, pp. 109–138. Wiley & Sons, Chichester (1995)
24. Eckhardt, D., Caglayan, A., Knight, J., Lee, L., McAllister, D., Vouk, M., Kelly, J.: An experimental evaluation of software redundancy as a strategy for improving reliability. IEEE Trans. on Software Engineering 17(7), 692–6702 (1991)
25. Eckhardt, D., Lee, L.: A theoretical basis of multiversion software subject to coincident errors. IEEE Trans. on Software Engineering SE-11, 1511–1517 (1985)
26. Elmendorf, W.: Fault-tolerant programming. In: 2nd IEEE Int. Symp. on Fault Tolerant Computing (FTCS-2), pp. 79–83. IEEE CS Press, Newton (1972)
27. Ghallab, M., Laruelle, H.: Representation and control in IxTeT, a temporal planner. In: 2nd Int. Conf. on Artificial Intelligence Planning Systems (AIPS 1994), pp. 61–67. AIAA Press, Chicago (1994)
28. Goldberg, A., Havelund, K., McGann, C.: Runtime verification for autonomous spacecraft software. In: IEEE Aerospace Conference, pp. 507–516 (2005)
29. Gray, J.: Why do computers stop and what can be done about it? In: 5th Symp. on Reliability in Distributed Software and Database Systems, pp. 3–12. IEEE CS Press, Los Angeles (1986)
30. Grnarov, A., Arlat, J., Avižienis, A.: On the performance of software fault-tolerant strategies. In: 10th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-10), pp. 251–253. IEEE CS Press, Kyoto (1980)
31. Hatton, L.: N-version design vs. one good version. IEEE Software 14(6), 71–76 (1997)
32. Hennebert, C., Guiho, G.: SACEM: A fault-tolerant system for train speed control. In: 23rd IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-23), pp. 624–628. IEEE CS Press, Toulouse (1993)
33. Howey, R., Long, D., Fox, M.: VAL: Automatic plan validation, continuous effects and mixed initiative planning using PDDL. In: 16th IEEE International Conference on Tools with Artificial Intelligence, pp. 294–301 (2004)
34. Huang, Y., Kintala, C., Kolettis, N., Fulton, N.: Software rejuvenation: Analysis, module and applications. In: 25th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-25), pp. 381–390. IEEE CS Press, Pasadena (1995)
35. ISO/IEC-15408: Common criteria for information technology security evaluation
36. Jaeger, E., Hardin, T.: A few remarks about formal development of secure systems. In: 11th IEEE High Assurance Systems Engineering Symposium (HASE), pp. 165–1174 (2008)

37. Jalote, P., Huang, Y., KIntala, C.: A framework for understanding and handling transient software failures. In: 2nd ISSAT Int. Conf. Reliability and Quality in Design, Orlando, FL, USA, pp. 231–237 (1995)
38. Kanoun, K.: Real-world design diversity: a case study on cost. IEEE Software 18(4), 29–233 (2001)
39. Kelly, J., Eckhardt Jr., D.E., Vouk, M., McAllister, D., Caglayan, A.: A large scale second generation experiment in multi-version software: description and early results. In: 18th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-18), pp. 9–14. IEEE CS Press, Los Alamitos (1988)
40. Khatib, L., Muscettola, N., Havelund, K.: Mapping temporal planning constraints into timed automata. In: 8th Int. Symp. on Temporal Representation and Reasoning (TIME 2001), pp. 21–27. IEEE, Cividale Del Friuli (2001)
41. Kim, K., Welch, H.: Distributed execution of recovery blocks: an approach to uniform treatment of hardware and software faults in real-time applications. IEEE Trans. on Computers 38(5), 626–636 (1989)
42. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, F., Derrin, P., Elkaduwe, F., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: sel4: Formal verification of an OS kernel. In: 22nd Symp. on Operating Systems Principles (SOSP), pp. 207–220. ACM, Big Sky (2009)
43. Knight, J., Leveson, N.: An experimental evaluation of the assumption of independence in multi-version programming. IEEE Trans. on Software Engineering SE-12(1), 96–109 (1986)
44. Laarouchi, Y., Deswarte, Y., Powell, D., Arlat, J., de Nadai, E.: Connecting commercial computers to avionics systems. In: IEEE/AIAA 28th Digital Avionics Systems Conference (DASC 2009), Orlando, FL, USA, pp. 6.D.1–6.D.9 (2009)
45. Lacombe, E., Nicomette, V., Deswarte, Y.: Enforcing kernel constraints by hardware-assisted virtualization. Journal in Computer Virology 7(1), 1–21 (2011)
46. Laprie, J.C.: Dependability: Basic concepts and associated terminology. Dependability : Basic Concepts and Terminology LAAS-CNRS, 7 Ave. Colonel Roche, 31077 Toulouse, France, p. 33 (1990)
47. Laprie, J.C., Arlat, J., Béounes, C., Kanoun, K.: Definition and analysis of hardware-and-software fault-tolerant architectures. Computer 23(7), 39–51 (1990)
48. Laprie, J.C., Arlat, J., Béounes, C., Kanoun, K.: Architectural issues in software fault tolerance. In: Lyu, M. (ed.) Software Fault Tolerance, pp. 47–78. Wiley & Sons, Chichester (1995)
49. Lardner, D.: Babbage's calculating engine. Edinburgh Review 59, 263–327 (1834)
50. Leveson, N., Cha, S., Knight, J., Shimeall, T.: The use of self checks and voting in software error detection: an empirical study. IEEE Transactions on Software Engineering 16(4), 432–4443 (1990)
51. Littlewood, B., Popov, P., Strigini, L.: Assessment of the reliability of fault-tolerant software: A bayesian approach. In: Koornneef, F., van der Meulen, M.J.P. (eds.) SAFECOMP 2000. LNCS, vol. 1943, pp. 294–308. Springer, Heidelberg (2000)
52. Lone Sang, F., Lacombe, É., Nicomette, V., Deswarte, Y.: Exploiting an I/OMMU vulnerability. In: 5th Int'l Conf. on Malicious and Unwanted Software (MALWARE), pp. 7–14 (2010)
53. Lussier, B., Gallien, M., Guiochet, J., Ingrand, F., Killijian, M.O., Powell, D.: Fault tolerant planning for critical robots. In: 37th Annual IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN 2007), pp. 144–153. IEEE CS Press, Edinburgh (2007)

54. Lussier, B., Gallien, M., Guiochet, J., Ingrand, F., Killijian, M.O., Powell, D.: Planning with diversified models for fault-tolerant robots. In: 17th. Int. Conf. on Automated Planning and Scheduling (ICAPS), pp. 216–223. AAAI, Providence (2007)
55. Lussier, B., Lampe, A., Chatila, R., Guiochet, J., Ingrand, F., Killijian, M.O., Powell, D.: Fault tolerance in autonomous systems: How and how much? In: 4th IARP - IEEE/RAS - EURON Joint Workshop on Technical Challenges for Dependable Robots in Human Environments, Nagoya, Japan (2005)
56. Meulen, M.J.P., van der Revilla, M.A.: The Effectiveness of Software Diversity in a Large Population of Programs. IEEE Trans. on Software Engineering 34(6), 753–764 (2008)
57. Migneault, G.E.: The cost of software fault tolerance. In: AGARD Symposium on Software Avionics, The Hague, The Netherlands, pp. 37/1–37/8 (1992)
58. Mongardi, G.: Dependable computing for railway control systems. In: 3rd IFIP Working Conf. on Dependable Computing for Critical Applications (DCCA-3), Palermo, Italy, pp. 255–273 (1993)
59. Muscettola, N., Dorais, G., Fry, C., Levinson, R., Plaunt, C.: IDEA: Planning at the core of autonomous reactive agents. In: 3rd Int. NASA Workshop on Planning and Scheduling for Space, Houston, TX, USA (2002)
60. Nguyen-Tuong, A., Evans, D., Knight, J., Cox, B., Davidson, J.: Security through redundant data diversity. In: IEEE/IFIP Int. Conf. on Dependable Systems and Networks, Ancorage, Alaska, USA, pp. 187–196 (2008)
61. Nicola, V., Goyal, A.: Modeling of correlated failures and community error recovery in multiversion software. IEEE Trans. on Software Engineering 16(3), 350–359 (1990)
62. Oh, N., Mitra, S., McCluskey, E.: ED4I: Error detection by diverse data and duplicated instructions. IEEE Trans. on Computers 51(2), 180–199 (2002)
63. Penix, J., Pecheur, C., Havelund, K.: Using model checking to validate AI planner domain models. In: 23rd Annual Software Engineering Workshop, NASA Goddard (1998)
64. Popov, P., Strigini, L.: Assessing asymmetric fault-tolerant software. In: 21st Int. Symp. on Software Reliability Engineering (ISSRE), pp. 41–450. IEEE CS Press, Los Alamitos (2010)
65. Randell, B.: System structure for software fault tolerance. IEEE Trans. on Software Engineering SE-1(2), 220–232 (1975)
66. Randell, B., Romanovsky, A., Rubira, C., Stroud, R., Wu, Z., Xu, J.: From recovery blocks to coordinated atomic actions. In: Randell, B., Laprie, J.C., Kopetz, H., Littlewood, B. (eds.) Predictably Dependable Computer Systems, pp. 87–101. Springer, Heidelberg (1995)
67. Scott, R., Gault, J., McAllister, D.: Fault-tolerant software reliability modeling. IEEE Trans. on Software Engineering SE-13(5), 582–592 (1987)
68. Smith, J., Nair, R.: Virtual Machines: Versatile Platforms for Systems and Processes. Morgan Kaufmann, San Francisco (2005)
69. Sullivan, G.F., Masson, G.M.: Certification trails for data structures. In: 21st IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-21), pp. 240–247. IEEE CS Press, Montreal (1991)
70. Tai, A., Meyer, J., Avižienis, A.: Performability enhancement of fault-tolerant software. IEEE Trans. on Reliability 42(2), 227–2237 (1993)
71. Tomek, L., Muppala, J., Trivedi, K.: Analyses using reward nets. In: Lyu, M. (ed.) Software Fault Tolerance, pp. 139–165. Wiley & Sons, Chichester (1995)

72. Traverse, P., Lacaze, I., Souyris, J.: Airbus fly-by-wire: A total approach to dependability. In: Jacquart, J. (ed.) Building the Information Society, 18th IFIP World Computer Congress, pp. 191–212. Kluwer Awademic Publishers, Dordrecht (2004)
73. Varian, M.: VM and the VM community: Past, present, and future (1997), http://web.me.com/melinda.varian/
74. Voges, U.: Software Diversity in Computerized Control Systems, vol. 2. Springer, Heidelberg (1988)
75. Xia, C., Lyu, M.: An empirical study on reliability modeling for diverse software systems. In: 15th Int. Symp. on Software Reliability Engineering (ISSRE), pp. 125–136 (2004)
76. Xia, C., Lyu, M., Vouk, M.: An experimental evaluation on reliability features of N-version programming. In: 16th Int. Symp. on Software Reliability Engineering, ISSRE, pp. 10pp.– 170 (2005)
77. Xu, J.: The $t/(n-1)$-diagnosability and its applications to fault tolerance. In: 21st IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-21), pp. 496–503. IEEE CS Press, Montreal (1991)
78. Yeh, Y.: Dependability of the 777 primary flight control system. In: Iyer, R., Morganti, M., Fuchs, W.K., Gligor, V. (eds.) Dependable Computing for Critical Applications (DCCA-5), pp. 3–17. IEEE CS Press, Los Alamitos (1998)