Independent Safety Systems for Autonomy : State of the Art and Future Directions

Étienne Baudin, Jean-Paul Blanquart, Jérémie Guiochet, David Powell

Independent Safety Systems for Autonomy
—
State of the Art and Future Directions

Étienne Baudin[(1)], Jean-Paul Blanquart[(2)], Jérémie Guiochet[(1)], David Powell[(1)]

(1) LAAS - CNRS
7 Avenue du colonel roche
31077 Toulouse Cedex 4, France

(2) EADS Astrium
31 rue des cosmonautes
31402 Toulouse Cedex 4, France

Technical report LAAS-CNRS N° 07710

Contents

1 Introduction

Computers are more and more involved in various automated systems, and software tends to replace human decision. This is particularly true for tasks in furthest or dangerous places that human cannot reach. Autonomy is also interesting in space exploration missions, which imply very long communication delays and prevent complex real-time control of the autonomous system from the Earth. For all these reasons, past and current research are focusing on giving a better autonomy of these system, which should be allowed to evolve in a complex and non a priori defined environment, free from a continuous external control.

However, actual autonomy capabilities rely on artificial intelligence techniques often known as efficient, but also non predictable, and sometimes non dependable. A contest has been organized in March 2004 by the DARPA[1] in the Mojave Desert, in which fifteen autonomous vehicles had to run over 228 Km. The best robots did not go beyond the twelfth kilometre. During the 2005 edition, 18 of 23 bots did not finish the race. This shown that, in spite of the work that is carried out to increase the functional abilities of the robot, another important challenge concerns the dependability of the autonomous systems. Among the various domains covered by the dependability, the safety is defined in [1] as the freedom from accidents or losses. To our concerns, we may sum up the safety as 1) preventing the autonomous systems from damaging its environment and 2) avoiding catastrophic decisions that would lead to the loss of the objectives.

This is particularly interesting as long as, in some cases, some simple safety measures enforced for the automatic systems (such as preventing people from entering in the working area of the system) cannot be actually enforceable without calling into question the main goals of the autonomous system. Consequently, in addition to the safety process dedicated to the design of the autonomous system, a safety monitoring may be useful to manage the safety during the operations. This safety is mainly affected by the following classes of hazards:

- the adverse situations met by the autonomous system;
- insufficient perception abilities, caused by faults affecting the sensors or by perceptual uncertainties;
- an erroneous control of the autonomous system, due to faults affecting the hardware and the physical devices of the system; and design or implementation faults of the control software.

In the previous DARPA Grand Challenge, the safety requirements consisted of a manned support vehicle and remote emergency stop capability using a stop safety radio supplied by the DARPA. Obviously, this kind of safety monitoring restricts the actual autonomous abilities of the system, that is why the on-board safety system should be designed to be automatic and independent from the monitored system, in order to protect the system from the hazards mentioned above. This implies the online verification of safety properties from an external viewpoint, i.e. the application of rules determined independently from the main system design, and executed by a subsystem with its own monitoring resources (to compute, perceive, and react). However, one of the remaining problem is how to specify the behaviour of the safety system, i.e., how to define the set of safety properties. Safety analyses often leads to corrective measures to be applied on the system. Our goal is a little bit more specific, since the corrective measures are previously partially defined (the implementation of an online safety system), and consists in the production of safety monitoring properties aimed at being executed by the safety system.

Since the safety of the autonomous system will rely on this safety rule determination process, it has to be efficient and clearly defined. In the literature, the monitoring systems are often described in term of architecture and design choices, but not really in term of

[1] http://www.darpa.mil/grandchallenge/

safety analysis. Consequently, our main goal is to link the design of the safety system with the upper part of its development process, including the safety analysis and the derivations of the safety rules. For this, a state of the art (complementary to [2]) about safety monitoring systems is presented in this article, which is aimed at presenting the explicit methods that are presented in order to specify the behaviour of the safety system. Furthermore, we studied what are the characteristics of the design of the safety systems and their relevant monitoring capabilities.

The first part of this article is dedicated to the autonomy, giving a definition and presenting examples of software architectures. Then, the second part deals with dependability of autonomous systems, and presents fault removal methods, such as verification and test, and online dependability methods that involve robustness and fault-tolerance mechanisms, among which we may distinguish safety mechanisms. The second section presents several examples of safety systems, some of them monitoring automatic systems and others autonomous systems. The last part is an orthogonal view of the examples, organized on two themes. On one hand architectures are compared, that leads to the definition on three generic patterns and on the other hand their safety development process (including safety rule determination) are presented.

2 Autonomous systems and their dependability

This section is aimed at introducing the *autonomous system* concept and at presenting the different dependability issues that are related to this kind of system. After a first part in which autonomy is defined and illustrated by examples of decisional architectures, we will address the main concerns about autonomous systems and dependability, including software verification, testing, and fault-tolerance.

2.1 Autonomy

Autonomy, in its most generic sense, is defined as "the ability to self-manage, to act or to govern without being controlled by others". This definition is insufficient for our purpose, and can be applied to "dumb" automatic drink distributors as well as to "intelligent robots" or deep space exploration probes. A more suitable definition for our purpose needs to focus on the latter type of systems.

2.1.1 Definition

According to the definitions given in [3] and [4], an *autonomous system* is able to reason and take decisions to reach given goals based on its current knowledge and its perception of the variable environment in which it evolves.

An autonomous system features decision capabilities, which implies:

- A planner, that schedules tasks to be performed to reach the given goal
- A procedural executive component, in charge of refining high level plans into elementary actions to be executed.
- Sensors and actuators, that allow the system to perceive its environment (localization, object recognizing) and to act on it.
- An autonomous system might also include some learning capabilities, in order to adapt the system behavior according to the situations it meets.

To sum up, an autonomous system possesses deliberative capabilities based on decisional mechanisms, in order to make the appropriate decisions. These decisions taken by an autonomous system cannot be calculated a priori because of the complexity and the variability of the environment in which the system is evolving. Instead, decisions have to be taken in real-time during operation. For this, a search is realized in a possibly very large state space. The search process is accelerated via the utilization of heuristics and should compute a solution that satisfies the system's goals. In contrast, *an automatic system* reacts simply, predictably, and according to a predefined behavior. This segregation between automatic and autonomous systems is related to the complexity of the task the system has to perform, and to the variability of the environment taken into account. The boundary between autonomous and automatic systems is not sharp: between a purely reactive automatic system and a totally autonomous deliberative system, a spectrum of systems may be considered. In the following parts of the paper, we consider a system to be autonomous if it needs a decision process exploring a space state in order to perform its mission. We focus on this feature of the autonomous systems since we are interested to address the relevant issues from a safety viewpoint.

Most autonomous systems implement the decision process through some sort of planner, which has to produce a plan according to the current state of the system and its current goals. A plan is a set of tasks with temporal relations, which may be total (Task1 $<$ Task2 $<$ Task3) or partial (Task1 $<$ Task2, Task1 $<$ Task3). Most planners are model-based, using a description of the systems' capabilities that have to be taken in account during planning (For example, the ability to move, to take photos etc.). Three criteria characterize decisional mechanisms [5]: *soundness*, which is the ability to produce correct

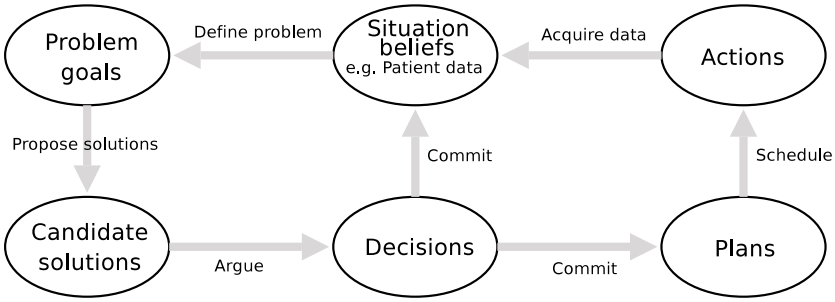

Figure 1: The Domino model (from [6])

results, *completeness*, which provides guarantees about the production of a solution and *tractability*, which characterize the complexity of the inference mechanism (polynomial, NP-complete). As these criteria are linked together, a tradeoff has to be made, for example soundness may be sacrified in order to ensure tractability.

2.1.2 Illustration

In this section, we present two models for autonomy. The first is the domino model, a high-level modeling of a reasoning process. Then, a more concrete example is given: a three-level software architecture for autonomous system control.

The Domino Model. This model was proposed in [6] to design an autonomous entity for the prescription of medical treatments (see figure 1). It models a generic decision process in a diagram, where the nodes are knowledge bases, and the arrows are deduction procedures that perform a step forward in the process.

This model is generic since it may be applied to various situations. The knowledge bases and inference procedures are not given in detail and have to be adequately implemented in the chosen context. It first starts from a knowledge base (Situation beliefs) that should lead to the definition of its main goals, via *problem definition*. This *problem definition* may be clearly translated in the medical domain since the considered prescription system first has to detect symptoms and to perform a diagnosis before treating the patient. This interpretation is harder to generalize to autonomous systems whose main goals are defined externally. However, some parts of the problem are not defined in the main goals: for example an obstacle in the environment that has to be perceived to avoid a collision. Once the current problem goals have been defined, a set of solutions is proposed (Candidate solutions), from which some decisions are taken (Decisions) after an argumentation. The decisions leads to new information in the situation beliefs and to the production of plan. Finally, this plan is decomposed into elementary actions (Actions) whose results are acquired in the situations beliefs.

The domino model, in spite of its abstract aspect, may be partially mapped to a software decision process for autonomous system control. For example we may recognize the search for a valid solution (Propose solution, Argumentation), the plan enactment (Schedule), the perception (Acquired date), etc.

The LAAS Architecture. This is a typical *hierarchical architecture* (see figure 2). Its layers are different in term of abstraction: highest layers deal with high-level perception and action whereas lower layers have local views and can only take specific actions. Reactivity is more important at the bottom of the architecture whereas the top layer needs to resort to complex decisional reasoning.

The LAAS Architecture is composed of three layers:

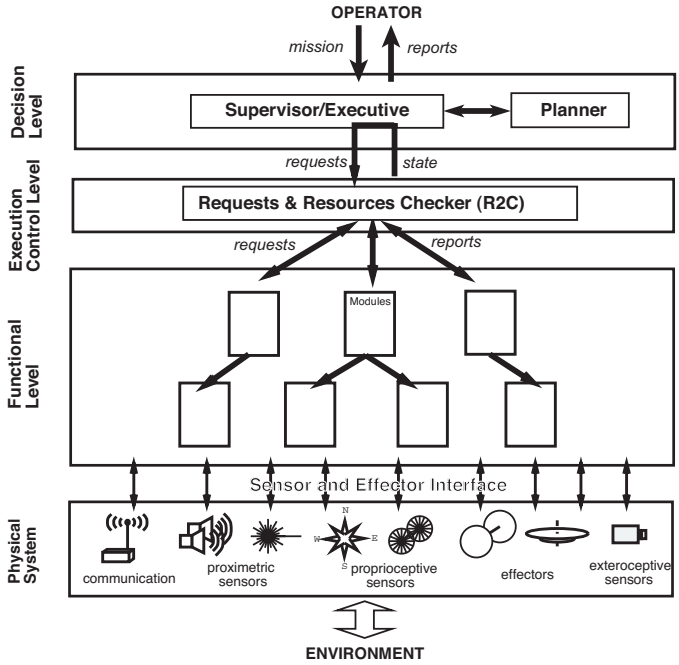

Figure 2: LAAS architecture for robot control

- A decisional layer including a temporal executive: the *IxTeT-eXeC* planner, and a procedural executive: *OpenPRS*. To produce a plan according to the goals, IxTeT-eXeC relies on a constraint satisfaction problem (CSP) solver using a model. This model is a set of constraints that describes the actual ways for the robot to reach its goals: its ability to move or to use its I/O devices. Searching for a solution has been accelerated in the solver with heuristics on the depth-first search. Even if the heuristic always leads the solver to a solution if it exists, it does not provide any guarantee about the time to find the solution. OpenPRS is in charge of the supervision of the plan execution. It processes high-level requests to obtain a set of low-level actions to be performed by the functional layer.
- An execution control layer, acting as a safety bag [7]. The R^2C component (Request & Report Checker) is a filter between the decisional and functional layers. On reception of a request from the decisional level, it checks the system safety and consistency and decides to commit the request, to reject it or to abort a currently running function that is inconsistent with the requested function. R^2C is detailed more in Section 3.2.1.
- A functional layer, aimed at managing low-level functions. It is composed of a set of functional modules, each one with a specific task. For example, one is trying to find paths, whereas another is managing electric motor control.

2.2 Hazards related to autonomous systems

Autonomy raises significant dependability challenges due to the high variability of the system environment. It also induces complex control software that is difficult to verify exhaustively, since the state space is potentially huge, and cannot even be defined in advance since some situations cannot be known in advance. For the same reasons, it is also difficult to evaluate how complete any test cases might be. Another threat to dependability is the utilization of heuristic methods to accelerate the search for a solution in the state space: this may reduce soundness or completeness but also introduces complexity that leads to a loss of predictability in the decisional mechanism that makes test and verification so

much harder.

The decisional layer is particularly challenging from a safety viewpoint since it produces and refines commands that can be potentially dangerous.

Taking inspiration from [8] and [9], we classify the main threats as follows:

- Endogenous hazards arising from the autonomous system itself:
    - Development faults, i.e., faults introduced during system design and implementation (by compromise, by accident, or even by malice).
    - Physical faults occurring in operation and affecting hardware resources (processors, sensors, actuators, energy sources).
- Exogenous hazards arising from the physical, logical and human environment of the autonomous system:
    - External faults such as physical interference, malicious attack, operator/user mistakes and inter-system cooperation faults.
    - Environment uncertainties due to imperfect perception of environment attributes (lack of observability, non-ideal sensors, etc.)[2].
    - Environment contingencies such as uncontrollable events, unforeseen circumstances, workload dynamics, etc.

Environment variability is taken into account by decisional mechanisms relying on artificial intelligence techniques such as expert systems, planning using constraint solvers, Bayesian networks, artificial neuron networks, model-based reasoning, etc., that take decisions from their perception of the environment, their self-perception and from the means they have to reach their goals. These decision processes include some form of domain-specific knowledge and a means for making inferences based on this knowledge. This kind of technique leads us to consider the following potential fault sources [9]:

- Internal faults of decisional mechanisms
    - Wrong, incomplete or inconsistent knowledge base.
    - Unsound inference.
    - Unforeseen contingencies, the knowledge base may be correct but reasoning based on it may break down when it is confronted with unusual situations that are not foreseen by the designer.
- Interface faults of decisional mechanisms
    - Ontological mismatch, when a mismatch occurs between the meaning of a term used within a component and the meaning of the same term when used outside the component (by its user or by another component).
    - When there is human interpretation, it may be incorrect; in particular due to overconfidence or on the contrary incredulousness, due to lack of information on the way the result has been produced.

For example, a typical ontological mismatch may occur between two layers of a hierarchical architecture. Two different layers exchange data that may have different semantics at each level, since their abstraction degrees are different. Homogeneity of semantics in the models of the different layers is a current challenge in robotic software architecture. For example, the IDEA architecture [10] was designed with agents using the same formalism to describe their models, in order to reduce inconsistency.

Internal errors can be illustrated by the domain model used by a planner to produce a plan according to the objectives. It is a set of facts and rules, which is one of the system's major knowledge sources. It is identified in [11] as an important factor that affects the

[2]Uncertainties may also be considered as a consequence of an insufficient design. However, since it is clearly related to the difficulty to forecast all the situations, we arbitrarily decided to associate it with exogenous threats.

soundness of results. Moreover, heuristics are used for generating plans, and their tuning is also important for the quality of the plans produced. These kind of highly complex systems are susceptible to the "butterfly effect" due to their high sensitivity to any modification of either model or heuristic.

2.3 Offline dependability methods

The first class of method that may be used to increase dependability of an autonomous system is offline, and is mainly composed of offline model checking and testing. Currently, others techniques such as theorem proving and static analysis seems not to be actually applied in the autonomous system domain. However, offline methods are an important area of current research, since fault removal enables the dependability of autonomous systems to be increased before and during their deployment.

2.3.1 Offline model checking

The *model checking* verification technique may be defined as an automated means to check a formal model of the system with respect to a set of behavioral properties, by exhaustive exploration of the state space. If the properties hold, the model checker informs the user that it succeeded, else a counter example of an execution in contradiction to the properties is given, and exploration ends. For complex systems, the exploration time and memory space may become huge, which is qualified as the *state explosion* limitation. This has been partially solved with *symbolic* model checking methods using compact representations of states, such as Binary Decision Diagrams (BDD), which manage states in sets rather than individually.

In addition to its automated component, model checking has the advantage of possible use early in the design process, as a model is only required for checking. For this reason, on one hand it is an improvement on testing, but on the other hand formal guarantees are provided only on the model and absolutely not on the actual implementation. This is particularly true when considering the difficulty of defining a sound model from the actual system. Even if the model is successfully checked, a doubt still exists about the modelled system.

Model checking considers two types of properties to be checked:

- *Safety properties* state that nothing bad should happen (e.g., no deadlock). This *logical safety* is specific to the model checking technique and is not relevant to the safety related to the dependability.
- *Liveness properties* state that something good must eventually happen (every client will eventually get the shared resource).

Model checkers often use *temporal logics* to express theses properties. These logics are extensions of propositional logics with temporal modalities [12]. Time is considered as discrete, and propositional formulas are evaluated at each step of a trajectory in the model's state space. Frequent modalities[3] are $<next>$ which states a formula holds in the next step, $<always>$, a formula holds in each state, $<until>$, a formula always holds until another starts to hold.

We can distinguish two classes of temporal logic:

- *Linear Temporal Logic* (LTL), which expresses path formulas. It considers a linear sequence (time steps) of logic formulas (for example representing events) and checks properties on it. A typical example of using LTL is analyzing a single execution trace (see fig. 3).

[3]It should be noted that past time temporal logics also exist. Expressive power seems to be the same, the choice being mainly directed by the kind of properties to be expressed.

$\Box Safe\_State$
All the steps of the execution must be safe

$Power\_TurnOn \rightarrow \bigcirc Power\_TurnOff$
Any ON signal must be followed by an OFF signal at the next execution step

Figure 3: LTL formula examples

$AG(Safe\_State)$
All states of all possible executions must be safe

$AG(EF(Possible\_recovery))$
All states of al possible execution must allow a state to be reached from which recovery is possible

Figure 4: CTL formula examples

- *Computational Tree Logic* (LTL) which expresses state formulas. Rather than considering a single execution, we may want to express properties on states, such as "in state X, some execution paths must lead to state Y". This kind of property (see fig. 4) does not only consider the future as a linear path but rather as several possible futures reachable from the current state.

Specifications in temporal logics are also used for on-line verification and online error detection (see R2C and MaCS examples in section 3), in which we are particularly interested.

Model checking is often applied on model of a subsystem or on a model of a subsystem component, since state explosion limitation does not allow the verification of too complex models. In the autonomous system domain, there are many models used and that are potentially critical. This is the case for the *Planner model*. As stated in Section 2.2, soundness of this kind of model is fundamental to the system's behavior, since the planner relies on it to find a way to reach its goals. Particularly, model consistency is critical and, in [11], a method is presented to apply model checking techniques. The Remote Agent HSTS planner takes domain models written in LISP-like syntax. These models describe relations between the different states of the system, each state being represented as a predicate. The example of model given in the article shows relations between states as temporal precedence constraints, where the robot has to go to the hallway when moving from kitchen to living room. Thus temporal predicates are defined: Before(Task1, Task2) (Task1 ends before Task2 starts), Contains(Task1, Task2) (Task2 starts after Task1 starts and ends before Task2 ends), etc. To verify the model, it has to be translated into an understandable model for the model checker. As HSTS models are similar to SMV[4] models, they can be translated to a transition system that SMV can process. In the example, the transition system uses the states of the robot's model and temporal constraints are translated into transitions. An application with a "hole fixer robot" is presented. The domain model, consisting of 65 temporal constraints, leads to the generation of a verifiable model with thousands of states. Properties state that "*An execution can lead to fix the hole*" or "*The robot can eventually reach a state where fixing is possible*". The first property checked to be true, but not the second one because with bad initial conditions (low battery), the mission could not be completed.

Another example concerns the *FDIR model* verification. Livingstone is the Remote Agent's model-based Fault Detection, Isolation and Recovery System (FDIR) [13]. A

[4]SMV, Symbolic Model Verifier, is the Carnegie Mellon University model checker, cf http://www.cs.cmu.edu/ modelcheck/.

Livingstone model describes the system's redundant architecture and allows the system to diagnose hardware faults and to trigger reconfiguration. As with the previous example, the hardest step is the translation of the Livingstone model into a model that can be processed by a model checker. Here, 45 days were required to obtain a verifiable model. After that, checking revealed five concurrency bugs, of which four were judged as critical by the development team and would probably not have been detected by regular testing.

It's important to note that model checking can also be applied to lower abstraction levels than automata. For example, *Java PathFinder* [14] can directly take as input Java bytecode, translating it to Promela (SPIN model checker input language). Java Pathfinder was used to verify *Mobot*, a robot aimed at following a line on a road. It is not an autonomous system as defined in 2.1.1, although the approach proposed in [15] is interesting because the whole system is checked by Java *PathFinder*. The checked model is composed of the Java control software, a model of the continuous system (with differential equations) and a model of the environment. Safety properties concern speed, and distance to the line, whereas liveness properties deal with robot mobility and accuracy after a fixed amount of time. Model-checking time and required memory space were respectively seven seconds and 5 MB. No counter-examples were mentioned in the article, which is partially explained by the oversimplification of the system model. However, the main goal was reach, i.e., the verification of a model of the complete system.

2.3.2 Testing

Testing is another important concern in autonomous systems. Contrary to the model-checking, testing can be carried out on real systems evolving in real environment. However, as this class of system is supposed to evolve in an open environment, the state space is possibly infinite. Consequently, testing has to be extremely intensive in order to cover the larger range of situations, which is the main issue of the autonomous system testing. According to [16], testing the real system with "high fidelity test beds" is also very long to perform, if the designer has to test a real mission scenario. For Deep Space One, only the nominal scenario testing was carried out due to the time and resources limitations for testing. However, it has been accelerated by simpler testbeds, for example by testing a subsystem in a simplified environment.

Another important issue is the definition of the oracle [17], aimed at checking if the outputs of the tested system are correct with respect to the inputs. This can be automated, for example in [18], a plan execution component is tested. This component takes a plan as input and generates an execution trace. A framework, mainly composed by a test case generator and an observer, is set up (see fig 5). Plans (input of the tested application) are generated exploiting the search capability of a model checker (part of the test-case generator) and, for each plan, some properties are automatically extracted. These properties mainly deal with precedence and timing constraints on the execution of the tasks defined in the input plan. Then, the oracle (Observer) is generated with respect to the properties, the plan is executed (by the Application), and finally the oracle checks is the trace generated by the execution is correct.

As previously said in Section 2.2, the components of an autonomous system are very sensitive to any modification. Consequently, non-regression testing, which has to ensure that the new version of the tested system is as good as the previous, may be useful. For example, in [19], since slight modification on the model may lead to an important behavior change, non-regression testing of the models in the software architecture is done. For each version of the model, some executions are launched and a trace verification tool (Eagle[5]) analyzes the generated logs according to generic and problem specific properties (depending

[5] http://osl.cs.uiuc.edu/ ksen/eagle/.

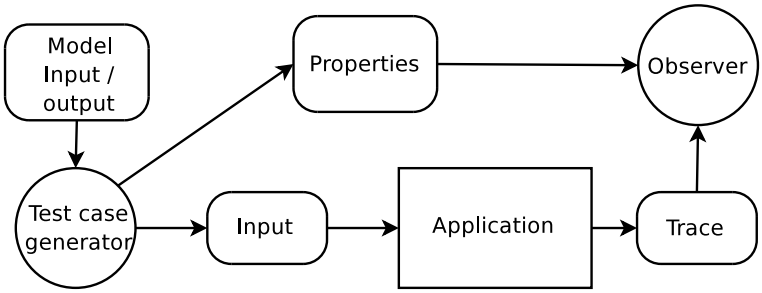

Figure 5: Test-case generator and trace verification framework (from [18])

on the model) that are not explicitly given.

This shows that testing the autonomous systems is an active area of research. It can be realistically performed, with real systems and real environments. In addition, oracles can be automatically generated. However, the problem of the limited coverage, because of the limited time and execution resources, still exists and is the main limitation.

2.3.3 The need for online techniques

Offline dependability techniques applied to autonomous systems can be:

- either formal on a model: we can obtain strong formal guarantees, but only on an abstraction of the system;
- or applied on a real system or software: accurate but incomplete, since all the situations that the autonomous system can met cannot be exhaustively tested.

Currently, event if these techniques are becoming more and more efficient and are essential to ensure a level of correctness of the design, they are not sufficient to guarantee a correct execution of the system during operations. For this, designers may use additional online techniques such as fault-tolerance and robustness.

2.4 Online dependability methods

Whereas offline dependability techniques aim to avert hazards that arise during development of an autonomous system (namely, development faults), online dependability techniques aim to handle hazards that arise during operation. This includes endogenous hazards such as physical faults and residual development faults (i.e., those not avoided by offline dependability techniques), as well as all classes of exogenous hazards (cf. Section 2.2).

2.4.1 Fault-tolerance and robustness techniques

As in [3], we classify online dependability techniques as either fault-tolerance techniques or robustness techniques according to the class of hazards being addressed

- Fault-tolerance techniques aim to avoid system failures in the presence faults affecting system resources (i.e., endogenous hazards such as sensor failures, software design faults, etc.)
- Robustness techniques aim to avoid system failures in the presence of external faults, environment uncertainties and contingencies (i.e., exogenous hazards).[6]

This distinction is useful since robustness issues have to be addressed mostly by the domain expert, whereas fault tolerance is the responsibility of the system architect.

[6]This definition is compatible with the term "robustness" used as an attribute for characterizing dependability with respect to external faults, as in [8], which did not consider other exogenous hazards.

Fault-tolerance techniques rely on system-level redundancy to implement error detection and system recovery [8], where an error is defined as "the part of the total state of the system that may lead to its subsequent service failure". System recovery consists of error handling (rollback, rollforward, compensation) and fault handling (fault diagnosis and isolation, and system reconfiguration and re-initialization). Error handling by rollback or rollforward is executed on demand, following error detection. Error handling by compensation may be executed on demand, or systematically, even in the absence of detected errors. This latter approach is often referred to as fault masking.

Since robustness techniques aim to deal with endogenous hazards, the notion of an error defined in terms of the system state alone is too restrictive. Instead, it is necessary to take into account the state of the system and its environment. Here, we use the term "adverse situation" to denote a global state of the autonomous system and its environment that may cause the autonomous system to fail to reach its goals.

Robustness techniques may be classified according to whether they handle adverse situations implicitly (by a treatment applied systematically in all situations) or explicitly (by a treatment applied on demand, following the detection of an adverse situation) [5].

Implicit handling of adverse situations may be likened to fault masking since the same treatment is applied in all situations, adverse or not. Examples include uncertainty management approaches such as fuzzy logic, Kalman filtering and Markov decision processes, and sense-reason-action techniques such as planning. In the latter approach, a search is carried out through projections of the currently-perceived situation towards possible future situations for a solution sequence of actions able to achieve the system's goals. The redundancy resulting from the combinations and permutations of possible actions increases the likelihood of a solution sequence enabling adverse situations in the possible futures to be circumvented. Moreover, if least commitment planning is employed, then adverse situations that arise dynamically while a plan is being executed will be easier to elude.

Explicit handling of adverse situations may be likened to error detection and system recovery in fault-tolerance: an adverse situation is detected either directly (by appropriate sensors, model-based diagnosis, situation recognition, etc.) or indirectly (through observation of action execution failures). In the latter case, system recovery may consist in backward recovery (action re-execution in the hope that the dynamic situation has evolved and is no longer adverse), but more commonly, recovery entails some application-specific rollforward approach such as re-planning, plan repair or modality switching.

2.4.2 Reliability and safety techniques

An orthogonal classification of online dependability techniques may be made in terms of the intended dependability goal: avoidance of failures in general (reliability and availability) or avoidance of catastrophic failures (safety).

When reliability/availability is the goal, the key issue is the presence of sufficient redundancy of resources, of function and of information to ensure continued service in the face of failed resources, functionality that is inadequate for the current situation, or imprecise or erroneous information. The redundancy may be exploited concurrently, as in fault masking or implicit handling of adverse situations, or on demand, following detection of an error or a manifestation of an exogenous hazard.

When safety is considered, detection of potential danger is the primary issue, be it due to an endogenous or an exogenous hazard. If a potential danger is detected, a safeguard procedure (such as shutdown) can be initiated. Such a safeguard procedure can be viewed as a specific form of forward recovery.

If reliability/availability and safety are simultaneously required (which is usually the case), a compromise must be made since any attempt to ensure continued operation (reliability/availability) in the face of hazards can only increase the likelihood of a current or

future hazard being mishandled and leading to catastrophic failure. Conversely, shutting down the system whenever danger is suspected (safety), evidently decreases the likelihood of continuous service delivery (the safest system is one that never does anything).

In a given system, reliability/availability and safety may be required interdependently at different levels. For example, the correct execution of a safeguard procedure (for system safety) requires continuous service (reliability/availability) of the mechanisms responsible for the safeguard procedure (e.g., the fly-by-wire system required to land an aircraft). As another example, for a system to be reliable, its subsystems need to be safe in the sense that they avoid failures that are catastrophic for the system as a whole.

A global approach to safety requires the consideration of both endogenous and exogenous hazards. So, although detection of errors due to internal faults can be seen as a necessary condition for a system to be safe, it is not sufficient alone – both external faults and the situation of the system with respect to its uncertain and dynamic environment must also be taken into account.

2.4.3 Independent safety systems

Since we decided to focus on safety, and since online dependability methods are required to ensure online correctness, the development of a safety system may increase significantly the dependability of the relevant safety-critical system. This is particularly true for the autonomous systems that use non "safety trustable" components.

The safety systems have to be independent, i.e., designed from an independent specification composed by a set of safety properties. The independence also concerns the online independence of the safety system, which has to enforce the safety properties independently from any faults of the functional system.

Examples are presented in section 3. Among these examples, we may distinguish Safety Bags and Safety Kernels. The Safety Bag concept is described in [20] and is defined as [21](C.3.4) "an external monitor implemented in an independent computer to a different specification". It monitors continuously the main computer and only deals with ensuring safety. Safety kernel is a little bit different in John Rushby's sense [22] it is a software monitor that isolates the control software from the features that may affect the safety of the system. Thus, when the control software asks the Safety Kernel to use a particular functionality, the latter can reject an unsafe command. For this, Rushby gave two conditions:

- Hazardous operations cannot be executed without being checked by the safety kernel, i.e., all safety-related operations must be observable at the kernel level.
- Safety properties should be expressed as a combination of operations observable at the kernel level.

For specifying the behavior of the safety kernel, as well as for Security Kernels, people often use the term "Safety Policy" to denote a set of high-level properties the kernel has to enforce, for example, the non-production of sudden movements.

Other expressions are used to denote a safety system: *Monitoring and Safing Unit* [23], *Protection System* [24], *Safety Manager* [25], *Safety Monitor* [26], *Checker* [7] and [27], *Guardian Agent* [6]. In the following sections of this document, we decided to use a generic expression to refer to the class containing theses systems: *independent safety system*, system in its most generic sense and independent as long as the safety system has to protect the system independently (with its own abilities whatever happens to the monitored system) from the functional system. We also use *safety system* to denote the same concept.

2.5 Conclusion

In this section, we saw what an autonomous system is, with a definition and an example of software architecture. Then, we introduced the hazards related to this kind of systems and the relevant offline dependability methods. On one hand, autonomous systems are difficult to verify exhaustively due to the variability of the environment, and the consequent state space to verify. On the other hand, testing is also carried out to reduce faults in the autonomous systems, but should be as intensive as possible according to the planning and the budget of the project. To complement these offline methods, on-line techniques are presented, to increase the reliability of the autonomous system, by robustness and fault-tolerant means. Finally, since the current online dependability techniques do not guarantee safety, we conclude on the necessity to use an independent safety system, only dedicated to the safety monitoring of the system, independently of its functional behavior. Thus, the next part of this text deals with examples of *independent safety systems*.

3 Examples of independent safety systems approaches

In this section, we present several protection mechanisms aimed at preventing a system from entering an unsafe state. All of them do not monitor autonomous system according to the definition given in the section 2.1.1, however they share a safety-critical feature. This section is divided in three parts. The first one deals with safety systems on automatic systems whereas the second concerns safety systems on autonomous system. The distinction is made since the safety systems may not have the same roles. For example, safety systems on autonomous systems often monitor the control software (see section 4.1.3) since the software decision process is critical for the system safety. The last part presents two generic frameworks for online verification.

3.1 Automatic applications

In this section, four systems are presented from various application domains: medicine, nuclear energy, railway and space. They all possess an independent safety system.

3.1.1 Magnetic Stereotaxis System

The Magnetic Stereotaxis System (MSS) [28] is a medical device for cerebral tumor treatment. A small magnetic seed is introduced inside the brain and is moved to the location of the tumor by magnetic fields produced by coils. The seed can be used for hyperthermia by radiofrequency heating, or for chemotherapy by delivering drugs at the location of the tumor. The seed is controlled by a human operator who observes movements on a display by magnetic resonance images. An automatic monitoring system uses X-Rays and a fluoroscopic screen to track the seed.

In [28], identified hazards, leading to potential patient injury are:

- Failure of electromagnets or controllers.
- Incorrect calculation of commands to provide a requested movement.
- Misrepresentation of the position of the seed on the MR images.
- Erroneous movement command by the human operator.
- X-Ray overdose.

The architecture is built using a Safety Kernel (cf. section 2.4.3) that filters commands and monitors the physical I/O devices (see figure 6) according to defined Safety Policies. Their safety policies seem to consist in safety rules, and we prefer to keep a higher level meaning of policy, that is a set of high-level properties.

The choice of using a Safety Kernel as an interface between the control software and the I/O devices is motivated by:

- Ensured enforcement of safety policies.
- Simplicity and verifiability of the Safety Kernel structure.
- Simplification of the application software, as the kernel frees the application of enforcing the policies.
- Kernel control of I/O devices.

Basically, the safety policies are grouped in class such as: hazardous operation device, device fails, erroneous input from computer, operator error, erroneous sensor input, etc., and concerns issues at the software level as well as at the system level.

3.1.2 SPIN

SPIN (French acronym for Digital Integrated Protection System) is a safety system for nuclear power plants [24]. It has been designed to prevent catastrophic situations such as meltdown that can lead to the spread of nuclear material outside of the plant. SPIN

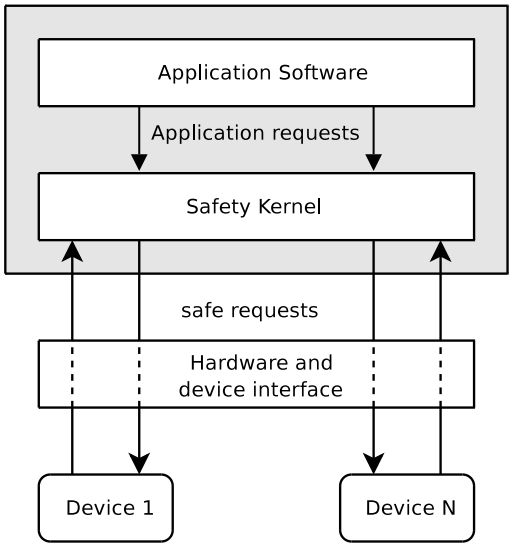

Figure 6: Magnetic Stereotaxis System's Safety Kernel (from [28])

monitors physical parameters such as the pressure of the steam or the temperature of the core. More precisely, SPIN ensures three groups of functions, categorized in [24] and [29] as:

- Protection functions, the monitoring of operational parameters
- Safeguard functions, the invocation of safeguard actions in case of accident, such as area islolation, activation of safety water circuit supply.
- Safety functions, aimed at switching to a safe position if a critical accident occurs, typically with the interruption the fusion reaction using the control rods.

SPIN's architecture is divided in two stages: acquisition and control (see figure 7). The first is composed of four *capture and processing units* (UATP). Each UATP includes two *capture units* (UA) in active redundancy and five *functional units* that execute the protection algorithms. UATPs do not communicate with each other, but only with the second stage, composed of two *safeguard logic units* (ULS). This control layer is in charge of performing safeguard actions and emergency procedures. Each ULS includes four *protection and processing units* (UTP), and each UTP receives signals coming from the four UATPs, and a vote is done to ensure the availability of the first stage. Then, a second vote (2/2) is carried out between each pair of UTPs inside each ULS. Finally, safeguards and safety actions are triggered after a last vote (1/2 for the safeguard actions, 2/4 for the safety actions).

Computers are monitored by local test units (not present on the figure) that are managed by a centralized test unit (UTC) that performs periodical checks. In case of an abnormal situation, a diagnostic procedure is triggered and performed by the *diagnostic unit* (UTD).

This monitor relies heavily on fault masking through functional redundancy and voting. This may be explained by the need for not only safety but also high reliability of the safety system itself, needed to increase the availability of the power plant.

3.1.3 Elektra

Elektra [20] is an electronic interlocking system for railways. As safety in transportation is a major issue, Alcatel Austria developed a safety system to prevent unsafe situations that could lead to disasters, such as collisions. Elektra is now in operation in many Austrian

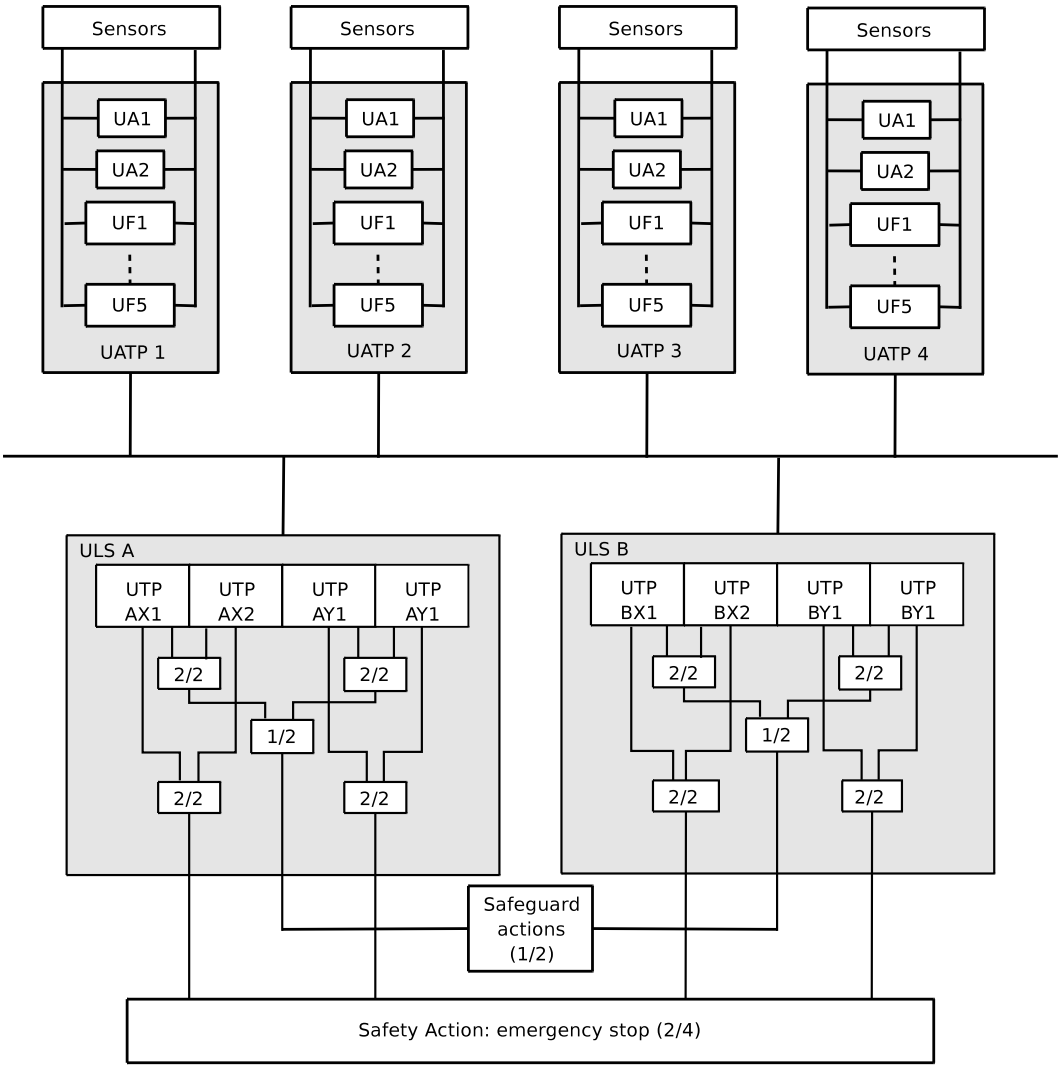

Figure 7: Architecture of the SPIN (from [24])

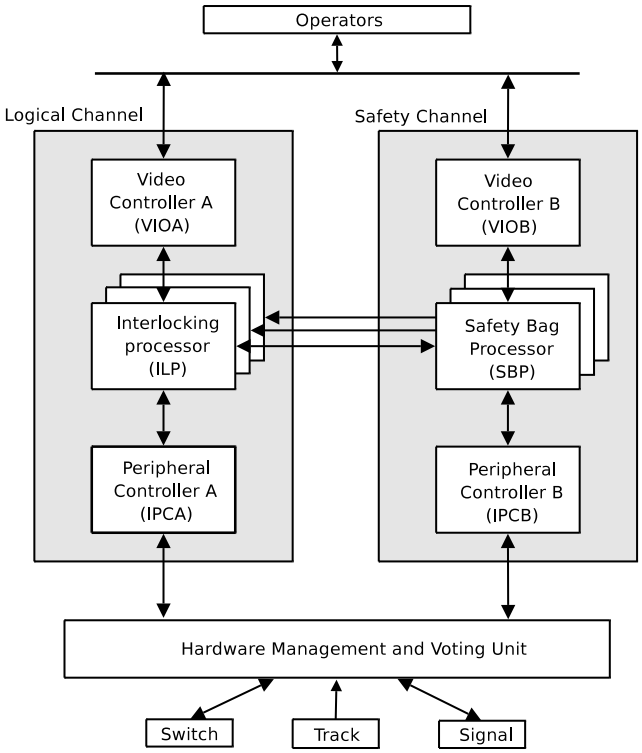

Figure 8: Architecture of the Elektra Safety Bag [2]

and Swiss railway sites. The safety requirements of Elektra are described by the *Austrian Federal Railway* standard, which states at most one catastrophic accident may occur within 10 years.

This is the first reference to the expression *Safety Bag* as a kind of diversification. A channel is dedicated to achieve the main function (functional channel), and another (safety channel) to ensure the safety of the decisions taken by the functional channel (see figure 8). The Safety Bag checks the commands produced by the functional channel, which are calculated from an order of the operator. If the command is not safe, the Safety Bag do not commit the command and the voting unit do not proceed to the execution of the command issued only from the functional channel. It uses its own expert system and knowledge base containing the current state of the system and the safety rules, and then decides whether the commands are safe or not. Safety rules are expressed in PAMELA, a logic programming language dedicated to the development of real-time control expert systems. The Safety Bag's redundant hardware, and communication between the safety and functional channel, are supported by the VOTRICS (VOting TRiple modular Computing System) layers to build a fault-tolerant safety channel in order to ensure availability in the system.

3.1.4 Automated Transfer Vehicle

The Automated Transfer Vehicle (ATV) is the European Space Agency autonomous spacecraft designed to supply the International Space Station (ISS) with propellant, air, water, payload experiments, etc. In the operational life of the ATV, the rendezvous and docking phases are hazardous due to the potential for collision with the ISS. To manage this kind of hazard, a Safety Unit has been developed to ensure the safety of the ATV during this phase. If a hazardous situation is detected, the rendezvous is aborted.

ATV's data management architecture is centralized, a single computer function handles

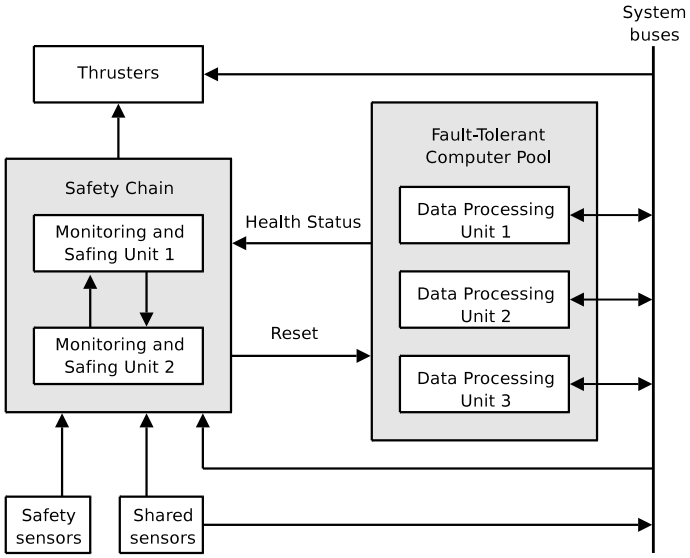

Figure 9: Automated Transfer Vehicle

all control tasks. This makes the system more predictable, easier to validate, and tends to limit the amount of software and relevant possible faults. Safety is addressed by a safety component (Monitoring and Safing Unit, MSU) to monitor the data management system by fault-tolerance. Safety and nominal systems are isolated in order to respect the following requirements ([23]):

- Operations shall be possible in case of nominal software failure.
- The MSU software data are directly coming from sensors.
- The MSU software shall have precedence over the nominal software. It must be able to passivate the nominal software.

The safety system (see figure 9) is composed by a pair of MSU and monitors a fault-tolerant pool of computers: the nominal system. This segregation pattern looks like Elektra Safety Bag (see section 3.1.3). The MSUs are also in charge of monitoring some ATV critical parameter such as attitude or distance to the docking port, and are executing the same safety software. The nominal system is composed by three computers, executing the same nominal software, which produce commands that are compared with a vote.

The safety system obtain the information about environment thanks to a fault-tolerant pools of sensors partially shared with the nominal system. However, the safety system uses raw data from the shared sensors whereas the nominal system uses processed data, thus the monitoring function is independent of any sensor-internal processing fault. If an hazardous situation is detected, the rendezvous is aborted: the functional computers are reseted and the safety chain takes the control of the ATV to engage the collision avoidance manoeuvres.

3.1.5 Ranger Robotic Satellite Servicer

The Ranger [26] is a robotic system aimed at refueling, repairing, and upgrading the International Space Station (ISS). It is composed of two seven degrees of freedom arms. This research project was lead by the University of Maryland and supported by the NASA.

Hazard analysis, including Impact Energy Analysis in order to analyze the system physics, highlighted the following catastrophic situations:

- Manipulator motion physically damages the Shuttle and prevents a safe return to Earth (e.g., by preventing the payload bay doors from closing).

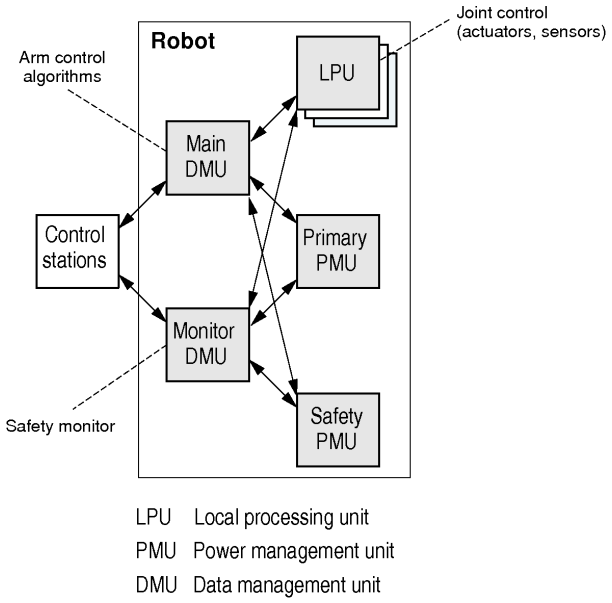

Figure 10: Architecture of the Ranger (from [26])

- Releasing an untethered object (e.g., an orbital replacement unit) that damages the shuttle or becomes orbital debris.
- Breaking an object due to excessive force or torque.

In addition, an Impact Energy Analysis has been performed in order to analyze the system physics.

The architecture is shown in figure 10. The production of safe commands is achieved by Data Management Units (DMUs). Main DMU produces commands and performs safety checking whereas Monitor DMU only performs the safety checks. The *Vehicle-Wide Safety Checks* concern:

- Position. Enforces the minimum approach distance determined by the impact energy analysis to prevent position violations.
- Velocity. Enforces the system-wide maximum velocity determined by the impact energy analysis.
- Inadvertent release of task equipment.
- Excessive force or torque of an interface.

All these checks are implemented in the same ways in both DMUs, except the one about position that was independently developed. As emphasized by the term *Vehicle-Wide* used in [26], the safety checks only concern external state of the system without taking into account internal data, such as hazardous commands.

The ranger has been developed, and was tested in laboratory and in water pool for hours in order to study the dexterous robotic on-orbit satellite servicing.

3.2 Autonomous applications

As well as for the automatic applications, we present safety systems for autonomous systems. The application domains are also various and concern space, robotics and medicine.

3.2.1 Request and Report Checker

As mentioned in section 2.1.2, R2C [7] is a safety and consistency checker located between the decisional and the functional layers of the LAAS architecture for robot control (see

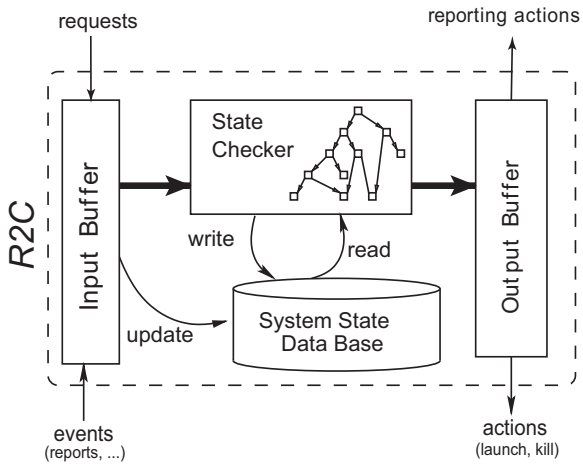

Figure 11: Request and Report Checker

figure 2, page 7). R2C is designed to avoid:

- Inconsistency of the system state between different abstract representations at different layers.
- Unforeseen incorrect behavior due to very high complexity of the decisional mechanisms.
- Incorrect low-level module interactions, since they execute concurrently and asynchronously.
- Adverse situations due to an open environment that cannot be exhaustively identified and therefore, tested.

R2C checks safety properties written in a subset of the CTL temporal logic. It takes as input requests produced by the decisional layer and reports coming back from the I/O devices through the functional layer (see figure 11). R2C is divided in two parts, a database that records the state of the modules, and the checker. The functional layer is composed of several modules, each one achieving a particular function and modeled by a state machine with *idle*, *running*, *failed*, *ended* states. The main goal of the R2C is to manage safety and consistency issues of these modules by sending them commands to change their state. This is done according to CTL[7] rules compiled into a controller relying on an Ordered Constraint Rule Diagram (OCRD)[8]. This controller is used online to detect potential violations of rules and to react to keep the system in a safe state.

As R2C is a controller, it does not really detect unsafe situations and afterwards trigger a recovery procedure. Rather, it continuously restricts the system's behavior by filtering each command that could lead to an unsafe or inconsistent operational situation.

3.2.2 Lancaster University Computerized Intelligent Excavator

Lancaster University Computerized Intelligent Excavator (LUCIE) [25, 30] is a commercial excavator that is controlled by an on-board computer that replace the human operator. The main goal is to have a system that reach the working area, dig trenches without the need of human intervention.

A *Safety Manager* has been developed to prevent accidents such as:

- Collision with an underground object.
- Collision with a surface object.

[7] Computational Tree Logic, see section 2.3.1.

[8] An extension of the Ordered Binary Decision Diagram with domain constraints.

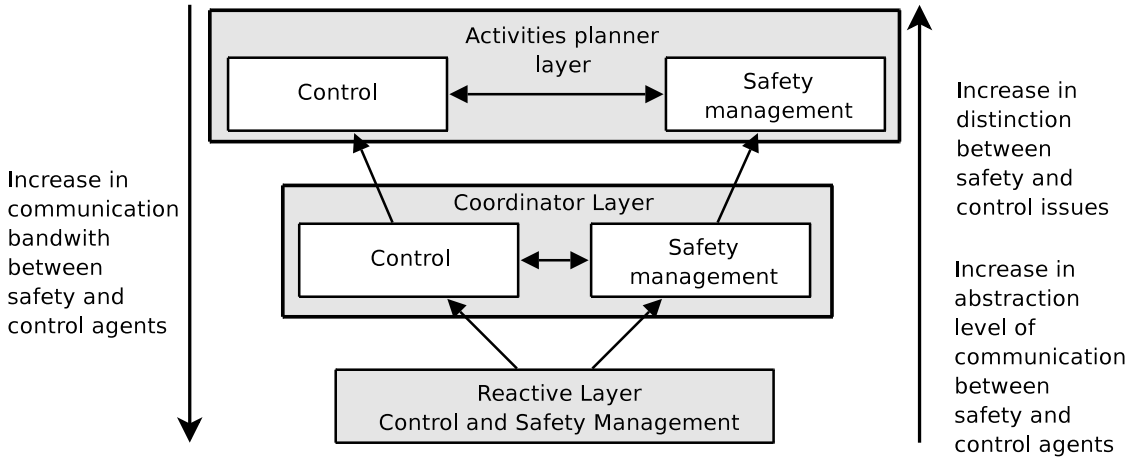

Figure 12: Safety component distribution in a three-layer architecture (from [25])

- Toppling of the excavator.

The control software of LUCIE is based on a three-level architecture for the control of autonomous system. It is similar to the LAAS architecture and is composed of an *Activities Planning layer* at its top, a *Reactivity-Coordinating level* and a *Low-Level Reactive Control layer* (see figure 12). The Safety Manager is distributed through the architecture, a component being present at each level.

In the Low-Level Reactive Control, safety is embedded in a module that also manages functional features. The Safety Module takes as input the desired command vector, modifies it make it safe, and then sends it to the excavator track and arm drives. It eliminates the hazards through real-time responses, for example an immediate motion of the track in case of tilting.

Safety is better separated from the functional components at the Reactivity-Coordination level. This layer decomposes high-level plans into subtasks, thus providing a low-level of reasoning. It operates on a local egocentric view of world. If a subtask is unsafe, the safety module of this layer can trigger a censoring action of the task. This safety component ensures safety from its local egocentric viewpoint, for example by avoiding a close obstacle.

At the Planning layer, an agent is dedicated to safety management. Communication between safety and functional planning agents is done through symbolic messages. Both agents use topological maps to obtain a global view of the environment and thereby facilitate moving of the excavator. They try to influence each other: on one hand the control agent tries to convince the safety agent that such or such an action is absolutely necessary; on the other hand, the safety agent tells the control agent that such or such an action is unsafe. The safety agent ensures that a consensus has been found and that it is safe. This higher-level safety component checks long term decision with a global view of the world, for example by checking the safety of the trajectory the functional software proposes. The distribution of the safety system through the architecture is more discussed later in the section 4.1.

3.2.3 SPAAS

SPAAS, Software Product Assurance for Autonomy on-board Spacecraft [31] is an ESA project carried out in collaboration between EADS-Astrium, Axlog and LAAS-CNRS. The main goal was to determine how to ensure safety and dependability for autonomous space systems. The need for autonomy in space is motivated by the following tasks:

- Continuous Earth observation

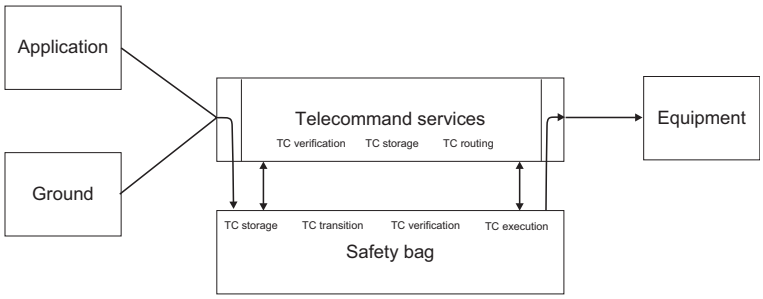

Figure 13: SPAAS Safety Bag

- Deep space exploration, with self-managing probes.
- Robotic services, such as repair missions or rock sample extraction, etc.

Two major threats possibly affecting safety were identified: adverse situations (exogenous) and software or hardware failures (endogenous). One of the recommendations of this study was the presence of a Safety Bag mechanism in the system, to keep it in a safe state. A prototype was developed and integrated in an existing software architecture. In this architecture, the messages coming from the ground, or sent from one on-board service to another, are managed by the Telecommand (TC) service. To ensure safety, all commands are checked by the Safety Bag. The checks are achieved by simulating the execution of the current command, and determining whether the induced state is safe or not. TCs were also previously checked by a Plausibility Checker before being sent from the ground. The Safety Bag acts as a TC service *wrapper*, although it is implemented as a normal service (see figure 13) due to constraints on the architecture. The Safety Bag behavior is described by hard-coded rules concerning on-board power: *global power level must not fall under a threshold*, and *every power-on command must be followed by a power-off one*.

3.2.4 Guardian Agent

Fox and Das, in their book *Safe and Sound* [6], presents artificial intelligence means to develop a medical prescription system, based on the concept of *intelligent and autonomous agent*. However, they also highlight specific threats related to this class of system (see section 2.2) and concludes to the need for means to increase the overall safety, since an incorrect prescription may lead to catastrophic situation such as the death of the patient. First, they designed the decision process on which relies each autonomous agent: the Domino Model (cf. section 2.1.2). The development of the agents is also facilitated by the utilization of a specific development lifecycle and a graphical language named *PROforma* to describe clinical decision process. In spite of these cautions, they conclude to the inability of the development process to foresee all of the hazards that can arise [32], and thereby to the utility of an online safety mechanism. Consequently, they developed an agent only dedicated to ensure the safety: the *Guardian Agent*.

Its concept was inspired by the Elektra Safety Bag, which was modeled in PROForma and was conceptually enriched on three main points:

- A higher level rule language. If-Then-Else rules do not explicit the rationale behind the rules. Thus, if a rule is inappropriate or inconsistent with the others, the system has no way of knowing it.
- The Safety Bag concept becomes generic, and not limited to a domain.
- Separation of the general safety knowledge and the safety expert domain knowledge.

As well as for the Domino Model that is not a software architecture, the Guardian Agent is not a precise safety system whose reaction are precisely defined in [6]. However, interesting concepts, such as generic safety rules to specify the Guardian Agent's behavior

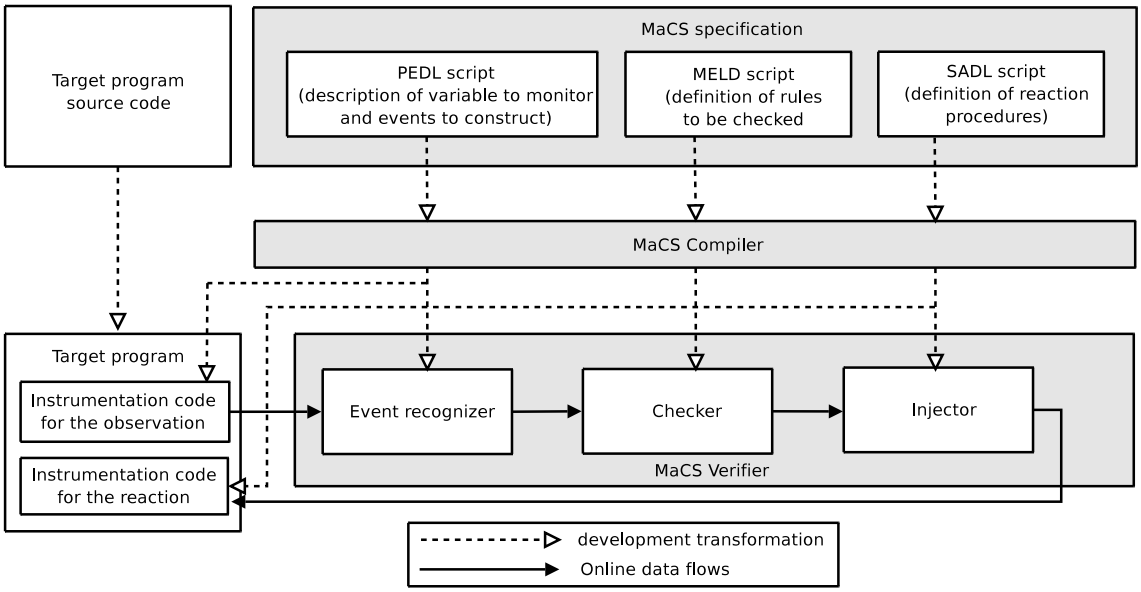

Figure 14: Monitoring Checking and Steering (MaCS) framework (derived from [27])

has been presented. Some of these aspects are revisited in section 4.2.3.

3.3 Generic frameworks

Independently from the application, generic frameworks exist for online verification. These frameworks formally verify properties on the current execution, and are a combination between testing and model checking. Among them, we may focus on two particular systems, which are actively developed and maintained.

3.3.1 Monitoring, Checking and Steering framework

The Monitoring, Checking and Steering framework (MaCS) [27] is a generic checker for software monitoring[33]. It is developed at the University of Pennsylvania and is being continuously upgraded. MaCS relies on event generation by the monitored software. These events are composed into high-level events on which properties are checked. The architecture of MaCS is shown on figure 14. The *filter* is the component "plugged" into the monitored software to extract data and transmit it to the event recognizer. It consists of two parts:

- the instrumentation of the Java bytecode, which is basically instruction insertion before or after a "key-point" (method call or return, attribute access);
- and the transmitter, a thread in charge of sending data through a socket.

The *event recognizer* builds high-level events from logic combinations of elementary events coming from the filter. These combinations are described in the *monitoring script* that is used for the automatic generation of the filter and the event recognizer. Then, events are checked by the checker according to a set of rules written in a script.

Finally, steering is performed through invocations by the checker of steering procedures defined in the *steering script*, which refers to methods already present in the monitored software. For steering, instrumentation is also required.

An application is presented with an inverted pendulum (IP, see figure 15). The device is controlled by a Java software including two controllers: a (trusted) safety controller, and an experimental controller to be tested. The goal is to swap from experimental controller to the safety controller if imbalance is detected, which is the single "safety criterion".

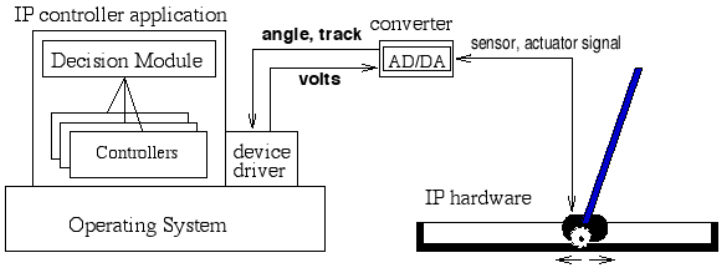

Figure 15: Inverted Pendulum (IP) device (from [27])

Low-level events are raised on the variation of internal variables of the program, such as the pendulum angle with the vertical axe, its position on the track, etc. Safety properties concern the possible range of positions of the pendulum, and the force to be applied to the cart, taking into account physical parameters. For the experiment, a faulty controller was developed from the safety controller, the faulty controller produces the nominal outputs until the moment when the output fails, taking a faulty constant value. During experiments, when outputs became incorrect, the checker successfully detected it and swapped to the safety controller. However, further analysis showed high latency between the failure of the experimental controller and the reaction during which 16 faulty values (320 ms) were sent. This was due to the checker being configured to send a large amount of data, with large buffers that must be full before actually sending the data through the network.

3.3.2 Monitoring Oriented Programming

MoP, Monitoring oriented Programming is a framework for advanced Design by Contract. It relies on code annotation in various logics, that are processed and either replaced by target code, or transformed into instrumentation code and a checker (as in MaCS, section 3.3.1). It is presented as "a light-weighted formal method to increase the reliability and dependability of a system by monitoring its requirements against its implementation". MoP is target language and logic independent. However the developed prototype, JavaMoP, works with Java and a set of various logics.

Annotations are extracted from the source code to obtain a logic specification, which is routed to the right logic engine. The logic engine processes annotations and produces abstract code. Abstract code makes the logic engines independent of the target language. Then, abstract code is translated into target code, which is finally integrated either directly into the source code, or compiled into an independent monitor fed with events produced after instrumenting the program using Aspect Oriented Programming to weave automata code into Java bytecode.

MoP comes with five logic engines: two Linear Temporal Logics (LTL, see section 2.3.1), past time LTL and future time LTL, regular expressions, Java Modeling Language (JML) and Java with assertions (Jass).

4 Analysis of examples

This section presents a transversal analysis of the safety systems presented in the previous section. The analysis is aimed at highlighting interesting features of each, and extracting the most appropriate ideas for developing an independent safety system for autonomous systems.

First, architectural issues are addressed, particularly concerning perception and action capabilities of the safety systems, classes of faults that are covered, and patterns of safety systems are proposed. Second, the process related to the determination of the safety rules is analyzed.

4.1 Architectural issues

From an architectural viewpoint, there are some important differences between the various safety systems Some of them act independently from the control software at a system level, while others are interfaced with it. The architectural differences are presented and analyzed in this section. First, an overview of the integration in the architecture of the perceptual and reaction capabilities of the safety systems is presented. Then, we discuss the impact of these design choices on fault coverage and on the abstraction level the safety system is working at. Finally, some architectural patterns are presented.

4.1.1 Observation and reaction levels

An independent safety system may be considered as a safety monitor of the considered system, and is supposed to observe the evolution of the system in its environment. From an external high-level viewpoint, safety is mostly related to the external state of the autonomous system, i.e., the state of its physical I/O devices (sensors, actuators, displays) and the state of some critical parameters (position, velocity), and the state of the close environment. Consequently, a safety monitoring should be efficient on the interface of the physical state of the system and its close environment. however, in addition to the physical evolution of the autonomous system, we may consider another dimension: its discrete evolution, related to the internal logical state of the autonomous system. This lead us to consider three levels of monitoring. The first one is the *internal level*, which monitors internal correctness of the data, such as commands, abstract representation, etc. The second monitoring level is the *functional external level*, which monitors directly the system's state using sensors of the functional system: 1) the health of the hardware components of the system and 2) the interface between I/O devices and the environment. The third one, *safety external level* also directly monitors these parameters but using safety sensors that are diversified from the funtional ones. These aspects affect the architecture of the monitoring system (developed in Section 4.1.4), but also the coverage of the safety system (see section 4.1.2).

In the table 1, we may notice that not all systems are directly monitoring the system using sensors. For example, R2C and the SPAAS Safety Bag only obtain their information indirectly via the control software. Concerning sensor data, some systems such as ATV or Ranger only use external information to manage safety. External state knowledge is the most pertinent for the immediate safety situation; however, checking control software data or produced commands may increase knowledge about safety, for example by analyzing plans or pending tasks to take into account what the system is expected to do in the future. This abstraction degree of monitoring is developed further in the Section 4.1.3. The safety external monitoring provides additional guarantees concerning on the sensed data, and is used in SPIN, which sense with its own sensors, and the spatial systems (ATV and Ranger). The ATV's Safety Unit has a part of its sensors that are shared with the

	Observation level		
	Internal	**Functional external**	**Safety external**
MSS	seed position within the brain	coil state	
SPIN			temperature safety sensors
Elektra	logical command checking	track and switch monit.	
ATV		Gyros	Sun sensors
MaCS	logical checking		
Ranger		Position sensors	?
LUCIE	Velocity, tilting management		
R2C	logical command checking		
SPAAS	TC checking		

Table 1: Observation levels

functional system, the other part being its own safety sensors.

It is important to note that this segregation is not directly related to the exogenous/endogenous one presented in 2.2. An endogenous hazard, such as a design fault leading to the production of an erroneous arm movement can be managed at the external level, whereas an exogenous hazard, such as an upset modifying a read only memory can be managed internally. However, the kind of data that is observed is dependent of the nature of the hazard. Most of the independent safety systems monitor parameters such as environment and physical behavior of the robot, that is mainly domain-dependent and covers exogenous hazards. However, monitoring the assurance of the abilities of the physical resources (processors, sensors, actuators, etc.) of the functional system to carry out their task with a sufficient level of correctness mainly concerns the system's architect, and covers the endogenous hazards.

After observation of the environment and the state of the controlled system, the safety system has to react when a safety rule is violated to put the system in a safe state. This reaction is a form of forward recovery and may be classified in two categories:

- Passivation reaction, when a safe state is directly accessible. This passive state may be reached by rejection of a permissive command[9] (Elektra), by switching off the actuators, etc.
- Active reaction, when the system has to perform an active task to reach a safe state. For example, when a hazardous situation is detected while it is trying to dock, the ATV engages a procedure to avoid collision with the ISS. When LUCIE starts to tilt, the balance has to be kept actively.

As with observation, reaction can be performed at different levels (see table 2). However, an important difference is that the environment can be observed, but not controlled. Reaction capabilities may be applied at different levels:

- Actuators, by resetting them or by removing power for example,
- Internal commands, for example, by rejecting an erroneous command that would lead to a hazardous situation, by reconfiguring computers or switching to a low power consumption mode.

Some systems, such as Ranger, ATV or SPIN, directly apply safety procedures to physical actuators, without considering the control software. Ranger just switches off the actuators, whereas ATV starts an active rendezvous abortion procedure.

[9] A permissive command let the receiver performing more hazardous actions than before.

	Level		Type	
	Actuators	**Internal commands**	**Active**	**Passive**
MSS	√	√		√
SPIN	√		√	
Elektra	√	√		√
ATV	√		√	
MaCS		√	√	√
Ranger	√			√
LUCIE			√	√
R2C		√		√
SPAAS		√		√

Table 2: Reaction levels and types

Other systems react both on actuators and by rejecting commands when necessary (Elektra, MSS) whereas the remaining systems (LUCIE, R2C, SPAAS) react only internally. They check whether commands to be sent to an output device or to another part of the system are safe and consistent with a given specification. Safety is only ensured at a local subsystem level, but not at the system level since nothing is done directly.

Concerning the health monitoring, the ATV Safety Unit monitors the health of the functional computers: the rendezvous phase with the ISS is initiated only if the number of healthy functional processors is sufficient to ensure the required safety level. In the MSS, one class of safety rules is only dedicated to check the health of the devices.

4.1.2 Hazards and fault coverage

The purpose of the safety system is to avoid hazardous events during the operational phases. Hazards are directly related to the physical abilities of the system and its ability to survive in its environment. These hazards can be the results of error that may be classified in two classes [22]:

- Omission errors, the expected functional action does not occur, and an active recovery procedure has to be initiated by the safety system.
- Commission error, the system applies a forbidden sequence of actions, and may be corrected by rejecting actions.

This is directly related to the application. For example, Elektra Safety Bag only deals with commission faults. It checks the safety of decisions about the affectation of a route to the trains, and can eject a command if it would lead to an hazardous situation. The safety bag has not to take active decisions since the correctness of the issued commands are sufficient to guarantee the safety of the routes. Some others systems, such as ATV Safety Unit has to take active decision to avoid hazardous events in case of omission faults.

Since only little information is given in the literature about the previous segregation on the autonomous systems, another interesting distinction between hazards may be done according to the origin of the hazards. According to Section 2.2, an unsafe behavior at the system level may be relevant to exogenous and endogenous hazards, respectively related to robustness and fault-tolerance. For each kind of hazard, the next parts of this section will deals about the main goals of the independent safety system:

- Protect the main system from hazards. For this, the safety system has to be independent from the functional system in order to detect them and to react adequately.
- Protect itself from hazards that may perturb a safe monitoring.

Exogenous hazards The primary goal of robustness is the tolerance of non-specified and adverse situations that may met the autonomous system. Functionaly, this problem seems complex to solve. However, from a safety viewpoint, it may be less difficult to define the boundaries of the safe situations whatever is functionaly specified. Thus, a safety system that correctly monitors the system can cover the adverse situations. More precisely, the efficiency of this coverage rely on the specification of the safety system, i.e., on the safety properties the safety system has been build on (cf Section 4.2). Another important issue robustness is uncertainties, in effect, the abilities of the system to perceive its environment are limited. Consequently, it should possibly make probabilistic evaluation to quantify the gap between what is perceived and what is considered by the system as the actual situation. In the presented systems, uncertainties on the perception of the functional system are not directly addressed by the safety systems. However, an unsafe behavior induced by uncertainties may be managed by the safety system, which reacts whatever is the origin of the error (adverse situation or uncertainty). This is also the case for the external faults. For example, MSS, Elektra and The Ranger safety systems cover operator faults. However, malicious attacks are evocated in none of the systems. A malicious attack on the functional system only may not affect safety since the safety systems would still be operational. the second class of robustness hazards of the exogenous hazards that affect directly the safety system itself. Environment uncertainties may also be a problem for the safety system. In LUCIE, this is partially managed with the utilization of stochastic models. Some kinds of external faults, such as space radiations that modify memories, are internally propagated an are managed as endogenous hazards with fault-tolerance techniques (see below). However, malicious attack against the safety system are not considered in the examples, since this kind of hazard is probably considered as currently unrealistic for these systems.

Endogenous hazards In addition to robustness, fault-tolerance has to avoid unsafe behaviors due to faults that affect internal resources of the system. These resources may be localized in the monitored system itself, or in the safety system. From a fault-tolerance viewpoint, the safety system has to:

- Protect the autonomous system from reaching an unsafe state because of faults that may affect the system. Monitoring the system from an external viewpoint may cover actuator and sensor faults, but also software and hardware ones, since they would induce unsafe behavior at the system level. In some cases, software faults may be considered as the main hazard (section 2.2 gives examples of typical faults in decisional mechanisms). This may also be covered by monitoring the whole system's functional behavior. Some health checks on components may also be performed. The safety system also have to be independent to the monitored system, i.e., it has to protect itself from the faults that it is supposed to cover. For example, the specification of the safety system has to be defined separately from the functional system design, to avoid common-mode failures. In addition, processor faults, can only be detected if the safety system is executed on its own processor. Concerning the devices, we may distinguish sensors and actuators. Faults in actuators may directly result in the production of an erroneous behavior, which has to be monitored whereas sensors faults may be handled later in the plan or act phases. We defined three levels of coverage for the actuators: the safety system acts through other software layers (*indirect* in the table 3), or acts directly with or without their own devices (resp. *direct* and *direct/own*).

- Protect itself from its residual faults, which could lead to lack of coverage of hazardous situations, or cause the system to leave a safe state (safety system fault tolerance).

	Fault coverage			
Coverage	Control software	Processors	Sensors	Actuators
MSS	√	√	√	direct/own
SPIN	√	√	√	direct/own
Elektra	√	√	?	direct
ATV	√	√	√	direct/own
MaCS	√			indirect
Ranger	√	√	√	direct/own
LUCIE	√			indirect
R2C	√			indirect
SPAAS	√			indirect

Table 3: Fault coverage in the examples

> Concerning hardware failures, the architecture of the safety system may use redundancy to perform a fail-safe continuous monitoring with reliability constraints or to provide self-checking and fail-stop operations. Software, actuator and sensor failures of the safety system induce the utilization of safety-critical methods for the development and adding fault-tolerant abilities to the safety system. For example, the software may be developed twice to introduce software redundancy. Sensor faults may also be covered by plausibility checks or detection and comparison of the results of redundant sensors.

In the table 3 we can see that all the systems cover software faults, since none of them are only hardware or health I/O device monitors. On one hand, some safety systems (R2C, SPAAS, LUCIE), directly monitor software, focusing their task on ensuring safety and consistency at the software level. This class of software monitor can also monitor actuators threw other software layer, but this management is only logical, i.e., only performed with software commands. This may cover some output devices faults but needs to trust 1) in the software interface layer between the safety system and the actuators, and 2) to trust the functional actuators used for the recovery. This is referred as the "indirect" output device coverage in the table 3. This kind of safety system is often used in autonomous software (see section 4.1.3). On the other hand, other safety systems (Elektra, MSS, ATV, Ranger) monitor the whole system state. Even if the software and its execution resources are not directly monitored, their relevant faults are indirectly covered. To carry out the monitoring, the monitor has to be itself independent from the faults that may affect the functional system, and concerning this we may distinguish few features. First of all, the safety system will not fail in case of functional hardware failure if it is executed on a separate computer. This is the case in SPIN, Elektra, ATV. For the Ranger, it is a little bit different since one monitor is on the functional computer and another is separate. MSS executes the safety system on a functional processor, however, an additional safety processor is in charged to detect any disturbance in the execution of the safety software. Logical independence is also required, i.e., the safety system has to be fully independent from any software module of the functional software, or any part of its data. For example, in MaCS the monitor is only fed with events coming from the monitored application and thus slaved to the monitored software. Concerning the sensors, only SPIN and ATV possess their own ones (ATV also shares sensors with the functional system) thus directly cover sensor faults. However, the MSS and Ranger safety systems share fault-tolerant pool of sensors with the functional system. With respect to actuator faults, MSS, SPIN, ATV and Ranger perform their recovery (actually a system or subsystem shutdown) with their own capability.

	Fault-Tolerance of the safety system			
Coverage	Monitoring software	Processors	Sensors	Actuators
MSS		√	√	
SPIN	√	√	√	?
Elektra		√	√	√
ATV		√	√	?
MaCS				
Ranger		√	√	
LUCIE	√			
R2C				
SPAAS				

Table 4: Fault-Tolerance of the examples of safety systems

According to the table 4, faults in the monitoring software are only covered in SPIN and partially in LUCIE. The LUCIE safety manager is also interesting since it is distributed in three components aimed at monitoring the three levels of the software architecture. As the levels are different representations of the same problem (the control of the system), monitoring at each level can be considered as a form of functional redundancy of the safety system. However, the Safety Manager becomes more complex, harder to validate and has to address the classic problem of internal consistency between its components (ontological mismatch, see section 2.2). Processors faults are covered with hardware redundancy in SPIN, Elektra, ATV, RANGER. The safety kernel of the MSS itself is monitored (heartbeats) by a little module setting the system in a safe state in case of no response from the safety kernel. In Elektra, if the processor of the safety channel fails, the safety bag will enter in a safe state blocking all the commands issued from the functional channel. Sensors are redundant in MSS, SPIN, ATV and Ranger (the MSS and Ranger safety systems share a fault-tolerant pool of computer with the functional system). In ATV, since most of the redundant sensors are used in cold redundancy, plausibility checks are performed in order to detect an error before switching to the redundant sensor. In Elektra, the operator interfaces are replicated.

To sum-up, the fault coverage of the safety system may be defined as a function of the following characteristics:

- Software in the safety system is specified and designed independently from the functional system (including its operator).
- End-to-end monitoring at the system level, to cover the maximum of faults.
- Execution of the safety software on a separate processor.
- Independent from the functional system, in order to protect itself from the faults the safety system has to cover. It takes its own decisions, it is executed on its own execution resources, and possesses its own perception and reaction devices.
- The safety system has to be itself safe and fault-tolerant.

4.1.3 Monitoring inside the software architecture

The independent safety system is aimed at preventing the monitored system from having unsafe interactions with its environment. This goal is the same for both automatic and autonomous systems. The difference in reaching this goal is that autonomous systems contain components working at a high level of perception and reasoning. Some safety systems take advantage of this high-level perception and reasoning to manage safety, which may explain why some safety systems only focus on software monitoring (cf table 1, page 28).

In addition to an "immediate" safety rule such as "The current velocity of the robot is too high considering that an obstacle is very close", we may want to express safety rules of a higher order, taking into consideration the "intentions" of the system. In the following part of this article, the intentions are referred as *projected actions*, in opposition to the *impending actions*. For example: "In the current situation, if the robot tries to reach this far site, its battery will run out of energy". This is a higher order monitoring in the sense that it does not simply monitor the current situation, but also what may be the future situation according to the projected actions of the autonomous system, in addition to the current state. If this safety rule had to be expressed with an "immediate" safety rule, the autonomous system would detect that it is entering in an unsafe situation just before reaching the point of non-return, too late to trigger a recovery procedure to go back to the battery recharging station. Thus, the choice of the abstraction level where the safety system is operating mainly depends of what kind of properties the safety system has to ensure. This is an example of how the safety properties (i.e., the specification of the safety system) may impact on the architecture of the safety system.

Analyzing the projected actions of the autonomous system in this way may provide beneficial features:

- Hazardous situations may be detected earlier.
- Reaction may be applied earlier, for example by rejection of a plan, or a pending task.
- Ensuring safety at the suitable level in the decision process, or at various levels.

However, monitoring functional software data induces an important need for synchronization between the functional software and the safety system. Moreover, This may reduce the independence of the safety system since it must cooperate closely with the control software, instead of working independently. This is the case in some examples: R2C, SPAAS safety bag, LUCIE. This reduction of independence is not obligatory, but often facilitates the design of such safety systems. It is important to notice that this kind of high-level monitoring increases the efficiency of the safety system, since hazardous situations are detected earlier.

Concerning software architecture, the LUCIE [25] example is very interesting. As LUCIE is a three-level architecture, designers decided to distribute the safety system throughout all the levels. For each level, there is a corresponding safety component that checks properties at the right abstraction level. But this approach seems complex, particularly in terms of consistency between the different safety components (see section 4.1.2). Moreover, there is no presentation of layer-dependent properties to substantiate this choice of safety system architecture. However, a layered safety system may allow the possibility of:

- checking the simultaneous consistency between the layers, by observing in the architecture, the same environment, goal descriptions, etc., from different viewpoints;
- tracking the safety of the decisions through the decision process.

Many of the safety systems (R2C, Elektra Safety Bag, MSS Safety Kernel) work in an intermediary mode. They do not really consider long-term projected actions of the system, but only the low-level command to be sent. If executing the command leads the system in an unsafe state, it is rejected. SPAAS safety bag has a more long-term view, since it checks in plans if a power-off command will be issued from the moment when a power-on command is sent.

In contrast, ATV and Ranger safety systems execute in a fully independent way from the main system. They only get information by sensors and act directly (if needed) on actuators. On the ATV, the only interaction between the safety system and the main system is an inhibition command from the former to the latter (since the rendezvous phase is short and the functional computers are tested at its beginning, the probability of the

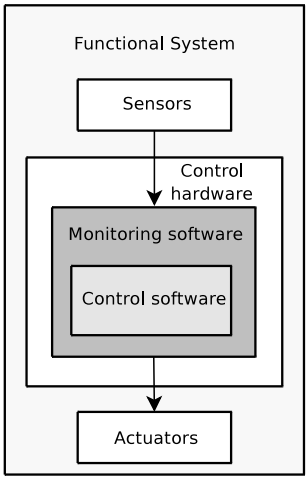

Figure 16: Software Monitor.

main software failing to receive the inhibition command is considered to be zero).

4.1.4 Architecture patterns

Considering the previous elements: perception and reaction capabilities of the system, the kind of faults that are covered, and the level of abstraction the safety system is working on, we can define three generic architectures (see table 5 and 6):

- Software Monitor
- Independent System Monitor
- Hybrid System Monitor

The *Software Monitor* (see figure 16) is a software module, often running on the same processor as the main software, which is aimed at monitoring the execution of the functional software. It can take various aspects: an active monitor running on an separate thread or process, a passive wrapper that reacts when it is fed by events coming from the control software, or inserted/weaved executable assertions. This safety system does not have its own capabilities of perception and action, thus relies on other piece of software. This approach is particularly used in autonomous systems (SPAAS, LUCIE, R2C) since the monitor can be integrated anywhere in the software architecture, and thus can be placed at the appropriate abstraction level. The monitoring should be positioned such that the semantics of the monitored data facilitates analysis, e.g., at the output of a module issuing high-level messages or commands. However, the main disadvantage is the high dependence of the safety system on the functional software. Figure 16.

An *Independent System Monitor* (see figure 17) executes on an independent computer, and has direct access to sensors and actuators or even its own ones. Interaction between the safety and functional software is restricted to the minimum, like an inhibition action when the safety monitor takes the control of the actuators. This safety system has a higher level of independence since it is monitoring the system from an external viewpoint, in an end-to-end way. The counterpart is that the projected actions of the autonomous system cannot be taken into account, causing safety violations to be detected at the latest possible time. Thus, the set of the safety properties checkable by such a system may be quite restricted. This approach seems to be well adapted when ensuring safety consists of a continuous monitoring with triggering of an active recovery procedure. As this class of monitor can "wraps" any kind of systems, it is not specific to autonomous systems and manages safety at a low abstraction level.

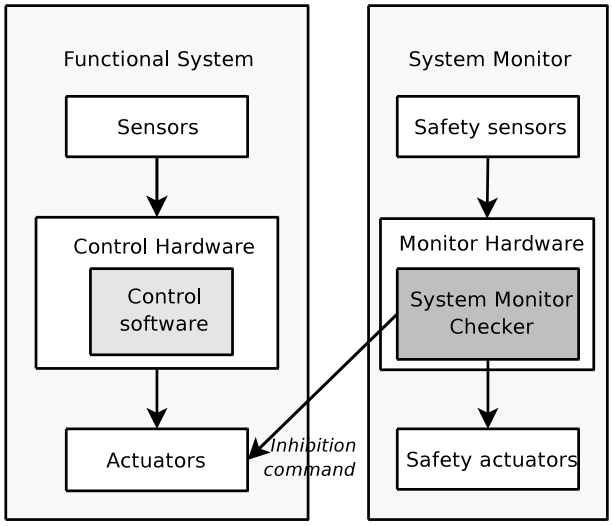

Figure 17: Independent System Monitor

The *Hybrid System Monitor*, is a combination between the first two architecture patterns (see fig 18). It gathers intention management and an end-to-end monitoring of the autonomous system. Intention may be observed or modified at various points in the architecture: at the planning level, between two communicating components or at the level of physical I/O. Instrumentation code has to be introduced to observe the system's projected actions and to perform censoring actions. This software module is driven by the main monitor. Censoring actions on the functional software may be complemented with compensation actions, to provide a reaction for the case of omission faults. However, it seems safer to act directly on the actuators, whenever that is possible. The counterpart of this kind of safety system is the difficulty of the design and its integration into the system, since it is both monitoring deep in the software architecture and at the highest level of the system to ensure the whole system safety.

Two examples, Elektra's Safety Bag and MSS's Safety Kernel, may fit to this pattern. They both analyze and stop commands before sending them to the peripherals. In addition, they manage safety in an end-to-end way since they directly perceive and act on the environment and the system. However, Elektra's Safety Bag needs to be synchronized with the functional channel twice: a) it must obtain the command to be checked; b) voting is carried out between the functional and safety modules to commit or reject it. MSS's Safety Kernel also performs command checking and continuously monitors physical variables (speed, location) of the seed placed in the brain of the patient. The LUCIE Safety Manager cannot be considered as a Hybrid System Monitor since it does not monitor the system level and does not ensures the independence of the safety system (it is deeply embedded in the software architecture).

In conclusion, the choice of the pattern is directed by two main factors. The first concerns the component that is the considered main hazard: is it the whole system or only the control software? If the software is the weakest part of a system, with other assumed dependable parts, a Software Monitor is the most appropriate approach, since it is the easiest to integrate in the system (for example as a thread or a process) and can monitor anywhere in the software architecture. On the contrary, if the whole system has to be monitored, the choice of the most appropriate safety system depends on the properties it has to enforce. If the properties only concern the current state of the system, then an Independent System Monitor should be sufficient. If not, the system's projected actions have to be managed and a Hybrid System Monitor is necessary.

	Patterns		
	Software Monitor	**Independent Monitor**	**Hybrid Monitor**
MSS			√
SPIN	√		
Elektra			√
ATV		√	
MaCS	√		
Ranger		√	
LUCIE	√		
R2C	√		
SPAAS	√		

Table 5: Mapping between the generic patterns and the actual systems

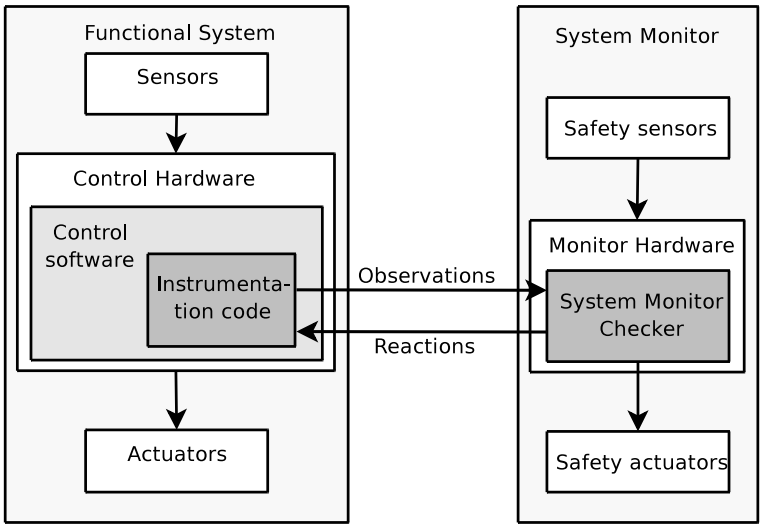

Figure 18: Hybrid System Monitor.

	Safety system pattern		
	Software Monitor	**Independent System Monitor**	**Hybrid System Monitor**
Monitoring and reaction level	control SW	System	control SW & System
Coverage of software design fault	√	√	√
Coverage of processor faults		√	√
Independent perception and reaction means		√	√
Decisional layer observation	√		√

Table 6: Characteristics of the patterns

4.2 Definition of safety rules

This subsection addresses the essential issues of hazard analysis, the derivation of safety properties, and their mapping to rules to be checked by a safety system. We call safety property a high-level domain constraint, in opposition to safety rule, which is a low-level executable design dependent constraint. The last part of this section deals with formalisms that can be used for expression of the safety rules.

4.2.1 Hazard identification

The hazard analysis process covers the whole development [1]. Its starts early with hazard identification, also known as Preliminary Hazard Analysis (PHA). It continues during the design, for example by modifying a weakness of the system or one of its subsystems. The process covers verification tasks during system development, and definition of safety systems and procedures for the operational life of the system.

Various methods may contribute to the identification of hazards. For example, Leveson [1] gives the following recommendations:

- Review pertinent historical safety experiences, lessons learned.
- Use published checklists and lists of hazards.
- Examine the basic energy source, high-energy items, hazardous material.
- Look at potential interface problems.
- Look at particular transition phases, etc.

Hazards are recorded in tables including the following information: concerned system, subsystem or unit; hazard description; hazard cause; effects on the system or its environment; hazard level; operational phases; corrective or preventive measures; verification methods; etc. In the examples, identification is often performed from *expert domain knowledge* that provides from experience a list of identified hazards. This was used in Elektra (railway is an old engineering domain and Austrian standards embody rules derived from experience), in SPAAS regarding rules about power consumption, and R2C where multiple unsafe interactions were identified in an "ad hoc" manner by researchers and engineers working on the system or one of its subsystems.

Other safety systems were designed using explicit methodologies to identify hazards. For examples, high-level hazards in MSS (events leading to a patient injury) were identified with a hazard analysis. For the Ranger, a Preliminary Hazard Analysis (PHA) was used to identify hazards related to manipulations: collision with the International Space Station, releasing of an object in space, breaking an object. For LUCIE's safety manager design, the first step was defining the basic functionalities that the excavator had to achieve. Then, according to these functionalities, hazards were identified in the most high-level manner in order to avoid omitting some situations. The second step consisted of a functional deconstruction to obtain a more precise view of the operational functionalities of the system, from which the relevant hazards (for example tilting during digging phase) were analyzed. This analysis is aimed at defining the dynamics of the hazards by looking at particular scenarios:

- Which sequences of system level events may induce the hazard?
- How may be best avoided the hazard? What are the remedial actions?
- If no remedial action on the system is possible, what are the necessary assumptions to ensure safety?

Moreover, other techniques allow these identified hazards to be refined according to the system design. We may cite Fault-Tree Analysis (FTA), a top-down approach that allows discovery of weakness of the system starting from an identified root event. Failure Mode Effects and Criticality Analysis (FMECA) is a bottom-up approach that starts from

the failure of a component and focuses on the actual effects on the system. Event-Tree Analysis is also a bottom-up approach, which considers a root event, and an ordered sequence of other events that determines the final state of the system. Cause-Consequence Diagrams are a combination of top-down and bottom-up approaches. Finally, HAZard and OPerability analysis (HAZOP) focuses the impact of the deviation of system variables on system safety, described with "guidewords" such as *more*, *as well as*, *reverse*, etc. HAZOP is mentioned in [6], in order to define the safety constraints to be enforced by the Guardian Agent.

4.2.2 Safety rule derivation

When the main hazards are identified, executable safety rules have to be obtained as a composition of the safety properties and safety analysis relevant to the system design, since the rules enforce properties on the actual designed system. The previously-identified safety properties are refined by dependability methods that rely on static or dynamic system decomposition. In the examples, only little information is given about the determination of the executable safety rules.

The most common mean to identify the properties is the fault-tree analysis, which lead to the deduction of the subsystem's role in the system safety. For example, fault trees were used in the ATV safing unit, in LUCIE's safety manager, and in MSS's safety kernel. In LUCIE, the use of fault trees was aimed at identifying internal system interactions from which conclusions could be drawn. However, the conclusions presented in [30] are high-level and mainly highlight some weaknesses of the subsystems: perceptual and reaction deficiencies, software and mechanical system integrity, inter-controller communication assurance.

Another use of the fault-trees is done in the MSS safety analysis. A generic fault tree was used to classify the safety rules according to their monitoring levels. At the root is the system failure, in the next level are subsystem failures, then for each subsystem may occur a physical actuator failure or erroneous input from software, etc. According to the available information, it only seems to help them in the classification of the safety rules, but not in their identification.

FMECA, jointly used with FTA were mentioned for the ATV and the Ranger safety system development, in order to help the design of both the system and the safety system.

Another step in the safety rule derivation, is the format of the final rule. For example, a rule may be either derivated to constraint between facts, such as in R2C, which can specify exclusive rules between services, or derivated to a detection-correction rule. The latter has two parts: a detection part, which specify a pattern to be recognized, and an action part, which specify what are the corresponding actions to be performed. This detection-correction process may be enriched in some ways. For example, a corrective action may not be safe in some cases, this also has to be checked. For this, an interesting approach for defining safety rules is that of the guardian agent. Fox and Das empirically defined a set of *generic safety rules* that have to be instantiated. Thereby rule determination is reduced to the extraction of domain safety data (required for the instantiation of the rules) since the global safety mechanisms previously exist (i.e., generic safety rules). However, no information is given about a method to carry out the determination of the safety data. The two first rules are given below:

```
/* Rule 1 */
/* Detect any potentially hazardous abnormality and */
/* raise a goal to deal with it. */
        if      result of enquiry is State and
                State is not safe
        then    goal is remedy State
```

```
/* Rule 2 */
/* If the abnormal state is a known hazard with a known */
/* remedial action then  */
/* propose it as a candidate solution to the goal. *//
        if      goal is remedy State and
                known remedy for State is Action
        then    candidate for remedy of state is Action
```

In order to facilitate the description of the facts that are managed by the generic rules, a dedicated modal logic *LSafe* (see section 4.2.3) has been developped with the relevant modalities.

4.2.3 Formalization of safety rules

In addition to their identification, or derivation, the safety rules have to be formally expressed in a language in order to be verified online. In this section, we present what kind of properties we may need to express, what languages are appropriate and which software mechanisms are necessary to implement them.

Simple rules may be expressed in propositional logic. Propositional logic is based on classic operators (and/or/xor/not) applied to boolean variables. A propositional expression only describes simple situations, such as the state of a system at time t. Propositional logic is often used with production rules: "if a pattern is detected, then trigger action". The expressive power of this class of the propositional logic is thus quite limited. However, all programming languages implement it and operators are even hardware-coded in processors. Elektra's Safety Bag rules were expressed with PAMELA (PAttern Matching Expert system LAnguage), a language for expert system design, based on propositional logic.

More complex rules can be represented in a temporal logic (see section 2.3.1). Temporal modalities express time relations between propositional formulas. In runtime verification, Linear Temporal Logic (LTL) is more common than Computational Tree Logic (CTL, see Section 2.3.1) because the linear aspect of LTL maps well to trace verification. Two families of LTL are available: past time LTL (ptLTL) and future time LTL (ftLTL). Their expressive power is quite similar and the choice between them depends on the context. As their names suggest, ptLTL expresses properties in the past (an acknowledgement is sent if a request has been previously sent) and ftLTL in the future (a opening parenthesis has to be followed by a closing parenthesis). LTL may be employed in a safety system to describe a dynamic behavior, where logical time is an important parameter. For example, past LTL is used in MaCS to write the monitoring script containing the safety rules. CTL is used in R2C, though the "tree" aspect of CTL is not really used. Past time LTL is simple to implement, as the value of a formula only depends on the past and present values of the boolean variables. On the contrary, future LTL formulas may be undefined since they require waiting for the occurrence of an event in the future. Büchi automata, alternating automata and binary trees (such as Binary Decision Diagram) are used to design safety monitors using temporal logic. Temporal logics are suitable to express "detection-correction" safety rules. For example, the system has to stop if an obstacle was previously detected may be expressed:

- ftLTL: $Obstacle\_Detected -> \bigcirc System\_Stop$ ($\bigcirc$ means "at the next step)
- ptLTL: $\odot Obstacle\_Detected -> System\_Stop$ ($\odot$ mean: "at the previous step).

However, nothing express what event is perceived, and what event is an action that has to be carried out. This may be included in the name of the event, but the formalism do not support this.

To express safety rules that include real-time boundaries, there exists extensions to temporal logic that introduce time stamps with the temporal modalities. For example,

MTL (Metric temporal logic) is an extension of ftLTL with time constraints. MaCS past LTL language has been enriched with time boundaries, as well as the temporal logic used in the Eagle tool, used for the test of a model-based planner (PLASMA) presented in section 2.3.2. Time-bounded properties can be implemented with automata such as timed automata.

The last example found in the literature is LSafe, a kind of deontic logic. Deontic logic is propositional logic with deontic modalities, which are often used to express a duty, in a context of morality or laws. This may be useful to express what the system should avoid (unsafe situation), whether forbidden situation has been reached, and how to return to a permitted state. LSafe contains the following modalities: `<Safe>`, `<Authorized>`, `<Permitted>`, `<Obligatory>`. LSafe only allows deontic facts to be written, which are then integrated in the generic safety rules (see section 4.2.2). In terms of expressive power, a property like "This unsafe situation must be avoided by performing one of these obligatory actions" is semantically more powerful than an "IF detection THEN action" production rule. Even if the resultant monitoring algorithm might be the same, the management of properties (for example for checking consistency between the rules) would be easier since the rules may carry more information.

4.3 Conclusions

Concerning the architecture of a safety system, two main factors have to be considered. The first is related to dependability and concerns fault-coverage. A safety system, in addition to the monitoring of the functional system, has to be executed independently such that is not affected by the faults in the monitored system. It also has to be safe with respect to its own faults. The other factor is more specific to autonomous systems and concerns the sharpness of the prevision of the unsafe behavior of an autonomous system. For example, it may detected in real-time by monitoring the current situation, detected slightly in advance by checking commands to be sent to the actuators, or detected earlier by the monitoring of the projected actions of the system. The challenge is thus to be able to monitor at the desired level(s) in the software architecture, while keeping a full coverage and full independence of the safety system. The desired level of monitoring is partially induced by the specification of the behavior of the safety system resulting from a dedicated development process.

In all of the examples, the development processes of the safety system are not precisely presented. Hazard analysis as well as rules and requirement derivation are only slightly mentioned, and are not the primary concerns of the papers presenting the safety systems. Thus, further work has to be done in the following directions:

- Hazard Analysis for autonomous systems. Extensions to existing approach are required since the interactions between the system and its environment are complex and difficult to foresee exhaustively. In addition to classic situations (for example the system runs too fast) that can be tackled by existing hazard analysis, new hazardous situations may emerge from the complexity of the relation between the system and its environment. To address this problem, classes of situations may be defined and a dynamic analysis may reveal unsafe scenarios. The obtained high-level properties may directly determine which level(s) of the software architecture need to be monitored.

- Hazard process dedicated to derivation of executable rules. since our goal is to specify a safety system, we have to adapt a classic hazard analysis process to a specific analysis only for corrective measures that can be implemented on an online independent safety system. This implies the restriction of the analysis to the steps that may produce executable safety rules, for example with the definition of a selection crite-

rion to filter executable safety properties among a set a safety corrective measures. This step may start early since it is partially design independent.

- A refinement framework that allows a formal specification of the safety system as a set of safety rules. This framework is aimed at 1) expressing formally the high-level domain safety properties; 2) deriving the safety properties according to document issued from the safety analysis process, such as FTA or FMECA, and according to a hierarchical description of the software architecture, to choose the right monitoring level in the decision process; and 3) performing some consistency checks, with techniques such as theorem proving or model-checking.

- Utilization of the underlying semantics of the languages of the framework to generate the software part of the safety system from its formal specification (mainly the set of safety rules).

5 Conclusion

After a survey of existing autonomous systems and their dependability features, we conclude on the necessity of an independent safety system to ensure the safety of critical autonomous systems. Indeed, since autonomous systems are expected to evolve in an open environment, the mechanisms involved in the perception, actuation and reasoning are an important source of hazards. The current dependability effort is mainly carried out with offline techniques such as model-checking or testing, which do not provide absolute guarantees about online correctness, notably in term of safety.

We have analyzed the architectures and the relevant development processes of several examples of automatic and autonomous applications. The first issue concerns their architecture, which may be characterized by two main factors: dependability and the desired fault-coverage, and the sharpness of the monitoring of the software architecture for autonomous system control. It led us to define observation and reaction levels. These levels, jointly used with an analyse of the hazard coverage of the example, allows us to define safety system patterns, which may be useful for the development of an independent safety system. Another issue concerns the development process of the safety system, and particularly its specification via the definition of executable safety rules. Only little information is given about techniques (such as FTA, FMECA, etc.) used to determine the safety rules, and more work is needed in this direction.

Consequently, our objective is the definition of a structured or partially-automated process, which could lead to the definition of executable safety rules to be verified online by the independent safety system. This requires the adaptation of existing safety hazard analysis techniques to obtain specific corrective measures (executable properties) and to cope with the complexity of the interactions between an autonomous system and its environment. Another interesting area is the design of a refinement framework that allows one to 1) choose the adapted formal language to express the desired properties to specify the rules, and 2) derive them using safety analysis techniques such as FTA or FMECA, in order to fit to the actual system. Finally, a safety monitor may be generated from this formal specification.

References

[1] N.G. Leveson. *Safeware - System safety and computers.* Addison-Wesley, 1995.

[2] J. Guiochet and D. Powell. Etude et analyse de systèmes indépendants de sécurité-innocuité de type safety bag. Technical Report 05551, LAAS-CNRS, 2006.

[3] B. Lussier, R. Chatila, F. Ingrand, M.O. Killijian, and D. Powell. On Fault Tolerance and Robustness in Autonomous Systems. In *Proceedings of the 3rd IARP-IEEE/RAS-EURON Joint Workshop on Technical Challenges for Dependable Robots in Human Environments*, 2004.

[4] R. Simmons, C. Pecheur, and G. Srinivasan. Towards Automatic Verification of Autonomous Systems. In *IEEE/RSJ International Conference on Intelligent Robots & Systems*, 2000.

[5] B. Lussier, A. Lampe, R. Chatila, J. Guiochet, F. Ingrand, M.-O. Killijian, and D. Powell. Fault Tolerance in Autonomous Systems: How and How Much? In *4th IARP-IEEE/RAS-EURON Joint Workshop on Technical Challenge for Dependable Robots in Human Environments*, 2005.

[6] J. Fox and S. Das. *Safe and sound - Artificial Intelligence in Hazardous Applications.* AAAI Press - The MIT Press, 2000.

[7] F. Py and F. Ingrand. Real-Time Execution Control for Autonomous Systems. In *Proceedings of the 2nd European Congress ERTS, Embedded Real Time Software*, 2004.

[8] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[9] D. Powell and P. Thévenod-Fosse. Dependability Issues in AI-based Autonomous Systems for Space Applications. In *Proc. of the 2nd IARP-IEEE/RAS joint workshop on Technical Challenge for Dependable Robots in Human Environments, Toulouse, France*, pages 163–177, October 2002.

[10] N. Muscettola, G. A. Dorais, C. Fry, R. Levinson, and C. Plaunt. IDEA : Planning at the Core of Autonomous Reactive Agents. In *Proceedings of the 3rd International NASA Workshop on Planning and Scheduling for Space*, 2002.

[11] J. Penix, C. Pecheur, and K. Havelund. Using Model Checking to Validate AI Planner Domain Models. In *Proc. SEL98: 23rd Annual Software Engineering Workshop, Greenbelt, MD*, 1998.

[12] A. Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[13] B. Williams and P. Nayak. A Model-based Approach to Reactive Self-configuring Systems. In *Workshop on Logic-Based Artificial Intelligence, Washington, DC, June 14–16, 1999*, 1999.

[14] K. Havelund and T. Pressburger. Model Checking Java Programs Using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 1998.

[15] S. Scherer, F. Lerda, and E.Clarke. Model Checking of Robotic Control Systems. In *Proceedings of the 8th International Symposium on Artificial Intelligence, Robotics and Automation in Space (iSAIRAS)*, September 2005.

[16] P. Nayak, D. Bernard, G. Dorais, E. Gamble Jr., B. Kanefsky, J. Kurien, W. Millar, N. Muscettola, K. Rajan, N. Rouquette, B. Smith, W. Taylor, and Y. Tung. Validating the DS1 Remote Agent Experiment. In *5th International Symposium on Artificial Intelligence, Robotics and Automation in Space*, 1999.

[17] M. Feather and B. Smith. Automatic generation of test oracles-from pilot studies to application. In *Automated Software Engineering*, pages 63–72, 1999.

[18] C. Artho, D. Drusinsky, H. Barringer, A. Goldberg, K. Havelund, S. Khurshid, M. Lowry, C. Pasareanu, G. Rosu, K. Sen, and W. Visser ans R. Washington. Combining Test Case Generation and Runtime Verification. In *Theoretical Computer Science 336*, pages 209–234, 2005.

[19] A. Goldberg, K. Havelund, and C. McGann. Runtime Verification for Autonomous Spacecraft Software. In *IEEE Aerospace Conference*, 2005.

[20] P. Klein. The Safety-Bag Expert System in the Electronic Railway Interlocking System Elektra. *Expert Systems with Applications*, 3:499–506, 1991.

[21] IEC 61508. Functional safety of electrical/electronic/programmable electronic safety-related systems. International Electrotechnical Commission, 2001.

[22] J. Rushby. Kernels for Safety? In T. Anderson, editor, *Safe and Secure Computing Systems*, pages 210–220. Blackwell Scientific, 1986.

[23] Extract from: *ATV - FMECA for the ATV avionics*, EADS-Astrium.

[24] D. Essame, J. Arlat, and D. Powell. Tolérance aux Fautes dans les Systèmes Critiques. Technical Report N°00151, LAAS-CNRS, 2000.

[25] Conrad Pace and Derek Seward. A Safety Integrated Architecture for an Autonomous Safety Excavator. In *International Symposium on Automation and Robotics in Construction*, 2000.

[26] S. Roderick, B. Roberts, E. Atkins, and D. Akin. The Ranger Robotic satellite servicer and its autonomous software-based safety system. *IEEE Intelligent Systems*, 19(5):12–19, 2004.

[27] M. Kim, I. Lee, U. Sammapun, J. Shin, and O. Sokolsky. Monitoring, Checking, and Steering of Real-Time Systems. In *2nd International Workshop on Run-time Verification*, 2002.

[28] K. Wika and J. Knight. A Safety Kernel Architecture. Technical Report CS-94-04, University of Virginia - Department of Computer Science, 18, 1994.

[29] SPIN, Protection des réacteurs nucléaires. Technical report, Merlin Gerin, 1993.

[30] D. Seward, C. Pace, R. Morrey, and I. Sommerville. Safety Analysis of an Autonomous Excavator Functionality. In *Reliability Engineering and System Safety 70(2000) 29-39*, 2000.

[31] JP. Blanquart, S. Fleury, M.Hernerk, C. Honvault, F. Ingrand, JC. Poncet, D. Powell, N. Strady-Lécubin, and P. Thévenod. Software Safety Supervision On-board Autonomous Spacecraft. In *ERTS'04*, 2004.

[32] J. Fox. Designing Safety into Medical Decisions and Clinical Processes. In *proceedings of SAFECOMP*, 2001.

[33] F. Chen and G. Rosu. Towards Monitoring-Oriented Programming: A Paradigm Combining Specification and Implementation. In *Workshop on Runtime Verification (RV'03)*, 2003.