HAL
open science

Fault Tolerance in Autonomous Systems: How and How Much?

Benjamin Lussier, Alexandre Lampe, Raja Chatila, Jérémie Guiochet, Félix Ingrand, Marc-Olivier Killijian, David Powell

To cite this version:

Benjamin Lussier, Alexandre Lampe, Raja Chatila, Jérémie Guiochet, Félix Ingrand, et al.. Fault Tolerance in Autonomous Systems: How and How Much?. 4th IARP - IEEE/RAS - EURON Joint Workshop on Technical Challenges for Dependable Robots in Human Environments (DRHE), 2005, Nagoya, Japan. hal-01292930

HAL Id: hal-01292930

https://hal.science/hal-01292930v1

Submitted on 24 Mar 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Fault Tolerance in Autonomous Systems: How and How Much?

Benjamin Lussier, Alexandre Lampe, Raja Chatila, Jérémie Guiochet,
Félix Ingrand, Marc-Olivier Killijian, David Powell
LAAS-CNRS
7 Avenue du Colonel Roche, F-31077 Toulouse Cedex 04, France
{*firstname.lastname*}*@laas.fr*

Autonomous systems are starting to appear in space exploration, elderly care and domestic service; they are particularly attractive for such applications because their advanced decisional mechanisms allow them to execute complex missions in uncertain environments. However, systems embedding such mechanisms simultaneously raise new concerns regarding their dependability. We aim in this paper to present these concerns and suggest possible ways to resolve them. We address dependability as a whole, but focus specifically on fault tolerance. We present some particularities of autonomous systems and discuss the dependability mechanisms that are currently employed. We then concentrate on the dependability concerns raised by decisional mechanisms and consider the introduction and assessment of appropriate fault tolerance mechanisms.

INTRODUCTION

Real world experiments in *autonomous systems* appear in such domains as space exploration, elderly care, museum tour guidance and personal service. These are critical fields of application, as a system failure may result in catastrophic consequences regarding human lives or in economic terms. Moreover, autonomous systems might be particularly dangerous as their actions are not directly controlled by a human operator. Thus arises a need for *dependability*, that is a justified trust that the system will appropriately perform its mission and not cause catastrophes. One of dependability means, fault tolerance, considers in particular that faults are inevitable in any complex system, and aims to make their consequences negligible. This paper studies autonomous system particularities regarding dependability, and discusses how to improve fault tolerance in such systems.

We first describe particularities of autonomous systems in comparison with "classic" computing systems, that is the notion of robustness (somewhat similar to fault tolerance) and the use of decisional mechanisms. The second section presents a detailed state of the art on dependability and robustness mechanisms in autonomous systems. We then give conclusions and recommendations for the development of autonomous systems, and present future directions: possible fault tolerant techniques for the decisional mechanisms of autonomous systems, and an experimental framework based on simulation and fault injection for assessing their efficiency.

I. PARTICULARITIES OF AUTONOMOUS SYSTEMS

This section presents the main particularities of autonomous systems in comparison with classic computing system. We first enunciate the distinction that we make between robustness and fault tolerance, and then introduce notions specific to decisional mechanisms.

A. Robustness and fault tolerance

The concept of robustness appeared in the robotic field as an answer to the large variability of execution contexts resulting from robot operation in an open environment. If we consider that faults are just another cause of uncertainty then robustness might be considered as a superset of fault tolerance. However, a useful distinction between robustness and fault tolerance can be made by restricting the use of the former to the tolerance of adverse situations *not* due to faults. We therefore adopt the following definitions in the context of autonomous systems (Figure 1) [24]:

- *Robustness* is the delivery of a correct service in adverse situations arising due to an uncertain system environment (such as an unexpected obstacle or a change in lighting condition affecting optical sensors).
- *Fault tolerance* is the delivery of a correct service despite faults affecting system resources (such as a flat tire, a sensor malfunction or a software fault).

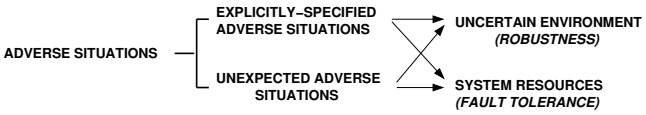

Figure 1: Robustness vs. fault tolerance

A remark may be added about the service provided by the system. Here, we consider that a *correct service* is a behavior in accordance with the intended system function, from a user point of view; it is defined *before execution* and is embodied in the system specifications, which determines the goals to be achieved in well-defined situations. A dependable system should thus provide a correct service regarding nominal situations and explicitly-specified adverse situations.

However, an autonomous system is often required to function in an open environment, where operating conditions can

not all be determined in advance. When faced with unexpected adverse situations a correct service cannot be guaranteed. However, the user may judge *after execution* that, given the unexpected situation that the system had to face, the service, while not correct with respect to the specification, was nevertheless *acceptable*. An acceptable service is generally less successful than a correct service as the system had to deal with unexpected adverse situations (it may achieve less goals, or only part of the goals).

B. Decisional Mechanisms

Decisional mechanisms are central to autonomous systems, as they embody the ability to select and choose actions to achieve specified objectives. This section presents decisional mechanisms and proposes a classification of possible faults affecting those mechanisms.

1) Characterization of decisional mechanisms: A decisional mechanism is composed of *knowledge* specific to the system's domain of application (such as heuristics or a model of the environment) and an *inference mechanism* used to solve problems by manipulating this knowledge. Ideally, the inference mechanism is independent from the application and can be re-used in a different one with new appropriate knowledge. In practice however, knowledge and inference mechanism are often hard to dissociate; for example heuristics in planners or weights in neural networks are intrinsic parts of the inference mechanism.

Applied to an ideal faultless knowledge base, an inference mechanism can be characterized by three main properties: soundness, completeness and tractability [29]:

- *Soundness* indicates that any conclusion raised by the inference mechanism is correct.
- *Completeness* indicates that the inference mechanism will eventually produce any true conclusion.
- *Tractability* characterizes the complexity of the inference mechanism: it indicates whether the inference mechanism can solve a problem in polynomial time and space, that is whether the time and memory space needed to find a solution can be defined as a polynomial function of the problem size.

Inference mechanisms currently used in autonomous systems are often sound and complete, but intractable. In practice, completeness is often relinquished through the use of heuristics, in order to improve tractability, and consequently the system performance.

2) Fault classification in decisional mechanisms: Decisional mechanisms introduce new faults in comparison to classic computing systems. From the risks of error presented in [5] and [13] we have identified *internal faults* in a decisional mechanism, and *interaction faults* between a decisional mechanism and another component or system (such as a human operator or another decisional mechanism).

a) Internal faults: Internal faults in a decisional mechanism may affect either the knowledge or the inference mechanism (Figure 2) [23].

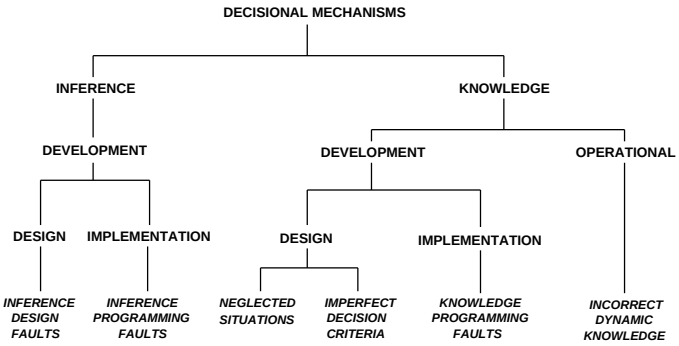

Figure 2: Internal faults in decisional mechanisms

Faults in the *inference mechanism* may be introduced during development of the system, either as a design fault (for example if the decisional mechanism is not adapted to the system function, or if its principle is flawed) or as a programming fault (such as a typing or algorithm error).

Faults in the system *knowledge* may be introduced either during development (design or programming) or in operation. Design faults may be either an explicitly-specified adverse situation that has not been covered by the developers (such as a missing procedure or an action needed to treat the adverse situation, or missing example sets used in learning for neural networks), or an imperfection in the choice criterion that possibly causes wrong conclusions to be drawn by the inference mechanism (such as faults in heuristics, or facts used for decision that are wrong in particular situations). Knowledge programming faults include both missing and faulty information in the knowledge of the decisional mechanism. Operational faults are incorrect dynamic information in the knowledge of the decisional mechanism, such as the current state of the system or information learned from the environment; these faults may be caused for example by sensor failures or imperfections, or undetected memory corruption.

b) Interaction faults: The main faults of decisional mechanisms due to interaction with other components or systems are *mismatches* in the exchanged information. Such faults are not specific to decisional mechanisms and can be classified in several categories [18].

Semantic mismatches are development faults that arise when different languages are used by the communicating components, and lead to information loss or modification. They are of particular concern in autonomous systems, as in practice semantic mismatches are common between different components. The three-layer type of architecture for example is built from components considering different levels of abstraction, each generally using a distinct model of the system and the environment. The executive component often lacks a global view of the system to better decompose plans, whereas the upper decisional mechanism does not possess all the relevant information for dealing with error reports.

Other types of mismatch (including physical, syntactical, and temporal mismatches) are similar to other computing systems, and can also be causes of information loss or modification.

At a different level, there are also risks of interactions faults between a decisional mechanism and a human operator:

- *Misguided optimism* in the system capabilities: the user is overconfident regarding the decisions taken by the decisional mechanism; for example, the results given may be very precise but wrong.
- *Incredulity* in the system capabilities: the user has no confidence regarding the decisions taken by the decisional mechanism; for example, since he does not understand the reasoning process leading to a given decision.

II. State of the art

This section presents a state of the art concerning dependability and robustness mechanisms in autonomous systems. We cover first fault prevention and fault removal (both encompassed in the notion of fault avoidance), then fault forecasting, fault tolerance and, finally, robustness.

A. Fault avoidance

Fault avoidance aims to produce a system containing as few faults as possible. It encompasses fault prevention, that is the means to prevent the introduction or occurrence of faults, and fault removal, that is the means to reduce the number and severity of faults.

1) Fault prevention: Fault prevention in autonomous systems is mainly implemented through the modularity of software components and the use of appropriate development tools.

Motivations for the *modularity of software components* are threefold: to decompose a complex system into simpler independent components, to capitalize generic structures or algorithm libraries, and to ease communication and scheduling between the components. Modularity thus prevent faults first by simplifying the development of each component (although it raises the problem of the component integration), and second by allowing components to be considered somewhat independently during development and testing. Modularity appears in generic architectures such as LAAS [1], RAX [28], CLARAty [34] or IDEA [27], but also in several components of these architectures.

Development tools generally include libraries of generic components, which can be instantiated to adapt to a particular piece of hardware or algorithm. They automatically generate part of the component code, accelerating the development process and reducing the risks of programming faults. Development tools are often used to develop functional layers for autonomous systems, such as the Genom modules of the LAAS architecture, or the ControlShell [32], ORCCAD [6] or SIGNAL[25] environments.

2) Fault Removal: Two fault removal mechanisms are particularly used in autonomous systems: tests and formal checking.

Simulation and operating *tests* are essential to the development of critical computing systems. In autonomous systems, however, their role is often limited to debugging rather than proving a thorough validation. Especially in the case of research platforms, developers check correct execution of the system for just a few situations. Intensive testing was however been carried out on the RAX architecture for the DS1 project [4]: six test beds were implemented throughout the development process, incorporating 600 tests. The authors of [4] underline the relevance of intensive testing, but acknowledge particular difficulties regarding autonomous systems, notably the problem of defining suitable test oracles.

Formal checking can be used to check properties of software component executions, such as safety or liveliness. However, it generally depends upon rigorous conditions, such as synchronism, and must be considered early in the development process. The ORCCAD and SIGNAL environments implement formal checking procedures.

B. Fault forecasting

Fault forecasting aims to estimate the present number, the future incidence, and the likely consequences of faults [3]. There are apparently few studies in fault forecasting for autonomous systems in the literature.

[7] presents studies on the failure of indoor and outdoor robots (some of which are not autonomous). Thirteen robots were observed for nearly two years, displaying an MTBF (*Mean Time Between Failure*) of about 8 hours, and a reliability of less than 50%. Outdoor robots were seen to fail more often than indoor ones (maybe because of the more demanding outdoor environment), and while hardware faults were the most common cause of failures (42%), the control systems (including the operating systems) were also significant sources of failures (29%).

[33] presents the implementation of the autonomous museum tour guide RoboX9 and a study of its failures during five months of operation. The MTBF reached and stayed at around 4.6 hours after four weeks of operation; 96% of failures where caused by the software components (80% due to the non-critical human interaction process, and 16% due to the critical navigation and localization process).

C. Fault tolerance

Fault tolerance is rarely explicitly mentioned in literature on autonomous systems. Although some techniques (such as temporal control by a watchdog, or positioning the system in a safe state after detection of an error) are quite common, we believe that their use is far from systematic, partly because most autonomous systems are still research platforms.

1) Error detection: Error detection in autonomous systems is mainly implemented by timing checks, reasonableness checks, safety-bag checks, and model-based diagnosis monitoring.

Timing checks are implemented by watchdogs in the RoboX9 autonomous system [33]. They supervise the liveness of critical functions such as speed monitoring, obstacle avoidance, bumpers and laser sensors. By their very principle, they detect only timing errors, not value errors.

Reasonableness checks verify, for example, that a state variable of the system lies within a specified interval of possible values. RoboX9 uses a reasonableness check to monitor the robot speed.

Safety-bag checks [20] consist in intercepting and blocking the system commands if they do not respect a set of safety properties specified during development. The R2C component of the LAAS architecture [30] acts as such a safety-bag.

Monitoring for *diagnosis* is mostly used to detect hardware faults (such as a flat tire, or an underspeed motor) rather than software faults. Diagnosis monitoring is implemented by checking system behavior against a mathematical model: for example, the MIR component of the RAX architecture detects the presence of faults by comparing its sensor values with theoretical values from the model.

2) System recovery: System recovery in autonomous systems is mainly implemented by error containment, positioning in a safe state, and hardware and software reconfiguration.

Error containment (to limit error propagation) is implemented in RoboX9 through the use of dedicated processors. One processor executes critical tasks (such as localization and navigation) whereas another one executes less fundamental and less thoroughly tested interaction tasks (such as the user interface and the face-tracking function).

Positioning in a safe state may be executed after a critical component failure or while executing a time-consumng recovery action, such as re-planning. For example, the RoboX9 system positions itself in a safe state when a critical subsystem fails, and the Care-O-Bot system [16] stops when a platform fault causes it to collide with an obstacle or person. Positioning in a safe state can also be executed after activation of a safety-bag, by blocking or halting activities which invalidate the safety properties.

Hardware reconfiguration is performed in the RAX architecture through functional diversification of the hardware components: when a hardware failure is detected, the MIR component seeks a new system configuration able to fulfill the functions of the faulty component.

Software reconfiguration can be executed either by switching between different execution modes or by corrective maintenance (which can be seen as an ultimate form of fault-tolerance) through the use of a patch. Switching between different execution modes may be realized through exception handling or through switching to an alternative decomposition of a task into elementary actions (see “modalities” below in section II-D.2.b). Patches have been implemented to recover the Martian rovers *Pathfinder* and *Sojourner* as they failed respectively because of priority and RAM management failures.

D. Robustness

Robustness in autonomous systems may be implemented in two ways:

- Systematic treatment applied in all situations, through observation of the current situation and selection of actions to be taken.
- Specific treatment applied only to adverse situations, through explicit detection of the adverse situation and triggering of an appropriate contingency procedure.

1) Observation and action selection: Systematic treatment by observation and action selection is similar to the fault tolerance technique of fault masking [3]. This approach exploits the massive redundancy resulting from the combinations and permutations of possible actions that can be carried out in any given situation. Thus, should an adverse situation prevent a subset of actions, it is likely that a sequence of alternate actions exists that allows the system to achieve its objectives. This approach to robustness is mainly implemented by planning and learning.

Planning gives an autonomous system the ability to select sequences of actions to achieve a set of objectives from an initial situation. It implicitly deals with environment uncertainties first by covering a large number of situations, which would be extremely fastidious to implement through imperative programming, and second, by tolerating situations that may not have been envisaged during system design, but which are compatible with its knowledge.

In CIRCA [15], this approach is taken one step further by taking into account prior knowledge about dangerous situations that may cause system failure. This prior knowledge is expressed as additional constraints input to the planner in order to ensure that the produced plans avoid such situations.

Least-commitment planning also aims to facilitate systematic treatment of adverse situations caused by environment uncertainties. It typically resolves a CSP (*Constraint Satisfaction Problem*) to minimally constrain the plan produced, and thereby provide temporal and resource flexibility during the plan execution. Adverse situations that were unknown while planning may thus be tolerated. The IxTeT planner from LAAS and the planner from RAX and IDEA implement least-commitment planning.

Similarly to planning, *learning* offers a systematic treatment of adverse situations by covering a larger number of situations than could be covered by explicit programming.

2) Detection and contingency procedure triggering: Specific treatment by detection and contingency procedure triggering is similar to the fault tolerance technique of error detection and recovery, as it aims to treat an adverse situation detected beforehand.

a) Detection: Detection is mainly implemented by execution control, situation recognition, and diagnosis.

Execution control supervises the execution of plans produced by the planner. A plan is generally developed under the condition that each of its actions will execute correctly; failure of an action thus indicates the appearance of an adverse situation that has not been taken into consideration by the planner. Execution control is the most used mechanism to detect adverse situations in the LAAS and CLARAty architectures.

Situation recognition uses an event chronicle to identify the current situation of the system and its environment among several generic models specified or learned during development. Work on situation recognition as a detection mechanism and its application may be found in [10] and [9].

Monitoring of the system for *diagnosis* is generally used to reveal physical faults affecting the system hardware. However, diagnosis mechanisms may also ascertain the situation of the system and its environment: in [26] the system learns a Markov Decision Process (MDP) to diagnose its situation, matching values of observable state variables to each type of possible situation.

b) Contingency procedure execution: Specific treatment of an adverse situation is achieved in planning through re-planning and plan repair, or in execution control through task retry and modality switching.

Re-planning consists in developing a new plan from the current situation and the objectives yet to be achieved by the system, explicitly taking into account the adverse circumstances that caused the failure of the previous plan. Positioning the system in a safe state may often precede a re-planning. Planners in the LAAS and CIRCA [15] architectures use re-planning.

Plan repair shares the same objectives as re-planning, but seeks a faster solution under the assumption that part of the previous plan may still be valid; execution of this valid partial plan is also executed at the same time as the plan repair. However, plan repair is sometimes impossible, so re-planning is required. The executive planner IxTeT-eXeC from the LAAS architecture [22] uses plan repair.

Since a high-level action requested by the planner can be decomposed into several alternative sequences of low-level tasks by the executive, *task retry* aims to select and execute another sequence of tasks whenever a previous one has failed, until either the action is successful or all decompositions have failed (in which case the action is reported as failed to the planner).

Modality switching extends the principle of task retry: each modality represents a possible decomposition of an action, particularly suited to a specific set of situations. Execution control selects the modality to be executed, and changes modalities when necessary (either when the action has failed, or when another set of situations has been detected). Modality switching may thus be carried out either when a change occurs in the environment (robustness), or when a modality fails due to some system malfunction (fault tolerance). [26] applies modality switching to robot localization and navigation; [31] presents switching between two navigation modes similar to modalities.

III. Towards Fault Tolerance and Dependability in Autonomous Systems

This section gives first some partial conclusions from our analysis of the current literature and some general recommendations concerning the development of critical autonomous systems. We then discuss the area on which we are focussing our attention.

A. State of the art analysis

Several conclusions can be drawn from the state of the art presented in the previous section.

1) Generally, we feel that the development of autonomous systems is lacking a holistic approach to dependability, through the combined use of its four means (fault prevention, fault removal, fault tolerance and fault forecasting). Currently, fault avoidance (i.e., fault prevention and removal) is largely privileged compared to fault acceptance (i.e., fault tolerance and forecasting).
2) When fault tolerance is considered, the focus is on hardware faults, especially faults that affect sensors and actuators. Nonetheless, fault forecasting studies show that software faults should also be taken into consideration.
3) Very few techniques address fault tolerance with respect to *development* faults. Robustess techniques contribute somewhat to it, but are surely insufficient for critical applications.
4) The autonomous systems community has yet to develop a measurement culture, both with respect to temporal performance and dependability: for example, we have found very few studies relating to fault forecasting of autonomous systems.

B. Recommendations for developing autonomous systems

This section presents conclusions and recommendations aiming to improve dependability in critical autonomous systems, adapted from [21].

1) An inference mechanism (the part of a decisional mechanism that is independent of the application) can be used for different applications, and problems encountered during its verification and validation are similar to those of “classic” software components. In practice however, inference mechanisms and knowledge are generally tightly linked, and it can be impossible to validate them separately. Moreover, validity may be complicated as autonomous systems have to take into account a large number of situations.
2) The use of learning mechanisms is not recommended for critical systems: first it is difficult to establish and ascertain their behavior as it emerges from examples; second, and in particular for online learning, the system behavior evolves without any direct control from the operator.
3) Although autonomous systems are supposed to operate with as little human intervention as possible, it is recommended to allow direct intervention by a human operator should the need arise (e.g., through remote control or an emergency stop switch).
4) A major challenge for development of autonomous systems is the occurrence of unexpected adverse situations. Robustness techniques can aim to ensure acceptable behavior but specific measures have to ascertain whether the system responses are safe and correct. Offline intensive simulation tests and online safety-bag mechanisms are thus strongly recommended.
5) Knowledge used by decisional mechanisms is a key factor in establishing the system behavior. In practice, the problems of correctness and completeness are complicated by the almost unavoidable presence of heuristics, which may trade correctness and completeness for tractability. We feel that more work is needed in this field, concerning first, the dependable formulation of knowledge and second, techniques for handling residual faults in the formulated knowledge.

C. Area of work

Our work aims to devise mechanisms for tolerating development faults in autonomous systems, an area which, as far as we know, is one that is yet to be explored. We focus on decisional mechanisms, and more particularly on their knowledge, as they represent the main distinction between autonomous systems and classic (non-autonomous) computing, for which efficient fault tolerance mechanisms have already been developed.

Two complementary aspects have to be considered: system safety and system reliability.

- The *safety* aspect aims to render the probability of a catastrophic failure negligible. Mechanisms for ensuring safety focus on the detection of errors that might violate safety constraints followed by positioning the system in a safe state.
- The *reliability* aspect aims to give correct or acceptable timely decisions, that is decisions that first, allow the system to provide a correct or acceptable service, and second, guarantee temporal constraints imposed by the system and environment dynamics. Ensuring reliability despite faults requires some form of error detection and service recovery.

Error detection has to be implemented in both cases. Covering a sufficient set of development faults requires the use of several complementary techniques: timing and execution checks, reasonableness checks, checks by diversified components, and, specifically for the safety aspect, safety-bag checks. Checks by diversified components appear to be difficult to implement in some decisional mechanisms (such as planners) since there are often a large number of valid possible plans, which cannot all be compared. Moreover, two equally valid plans may be strongly dissimilar, making it difficult to define a satisfactory choice criterion. Nonetheless, an automated generation of plan oracles has been successfully implemented during the validation of the RAX system [11].

Concerning the safety aspect, the use of a safety-bag to detect dangerous situations and and position the system in a safe state is attractive for many critical systems. We believe that safety-bags are a promising technique for autonomous systems, since their efficiency is not affected by uncertainties in the system behavior due to the use of decisional mechanisms. Nonetheless, work remains to be done on the *expression* of safety constraints to be dynamically verified, the actions needed to *position the system in a safe state*, and possibly more complex *reaction capabilities*.

Concerning the reliability aspect, two cases can be distinguished: the acceptability of decisions, and their timeliness. The *acceptability of decisions* can be disrupted by compromises in development, faults in decisional mechanisms, or environment changes[1]; we think that knowledge diversification, either in the heuristics or in the domain-specific model of a decisional mechanism, is a potential fault tolerance mechanism for addressing this concern. To improve the *timeliness of decisions*, possible fault tolerant techniques include concurrent use of different heuristics, or a relevant selection according to the type of problem addressed by the system.

[1]According to concepts presented in the first part of this article, problems raised by environment changes relate to robustness rather than fault tolerance.

IV. Future Directions

We are currently focussing on fault tolerance techniques addressing the reliability aspect of decisional mechanisms. We present below some possible alternatives for implementing fault tolerance in decisional mechanisms and then propose an experimental framework for assessing their efficiency. We are currently implementing this framework, prior to prototyping some of the evoked mechanisms.

A. Fault tolerance capabilities for decisional mechanisms

This section presents several possible research paths that we have identified for improving fault tolerance in decisional mechanisms.

1) Agent-based type of architecture: Although the "three-layer" type of architecture [14] is by far preponderant today in the development of complex autonomous systems, the *agent-based type of architecture* proposed in IDEA [27] offers a promising alternative. It actually considers an autonomous system as a group of multiple "agents", each possessing deliberative mechanisms and the same symbolic representation, and implementing a different system function (Figure 3). Each agent is capable of making independent decisions and taking actions to satisfy local goals based on requests from other agents and its perceived environment.

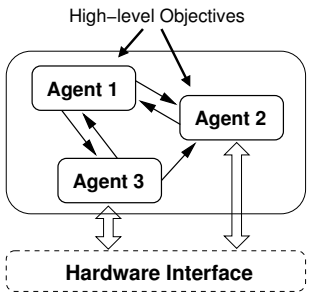

Figure 3: Agent-based type of architecture: IDEA principle

In comparison with more classical architectures, we think that several aspects of the agent-based type of architecture may improve system dependability and robustness:

- The same symbolic representation is used by the planning and executive components of all agents, thus limiting the risks of semantic mismatch (particularly model mismatch) between planning and execution control (fault avoidance).
- The flexibility of modular agent-based programming may allow the system to correctly react to a larger set of adverse situations than other more monolithic architectures.
- The system is explicitly divided into task-specific components (the agents), which may diminish error propagation and facilitate error recovery (fault tolerance).

2) Plan analysis: As a decisional mechanism may generate erroneous plans (for example because of residual development faults in the model), it is interesting to check on-line the validity of a plan before its execution, that is whether it contains inconsistencies, and possibly whether it advances the system towards achieving its goals. Such an error detection mechanism could trigger some form of forward recovery. It might be possible to develop an on-line acceptance test based on the previously mentioned work by Feather and Smith on oracles for off-line testing of the correctness of plans produced by the DS1 planner [11].

3) Heuristic diversification: As previously stated, heuristics are used in decisional mechanisms to deal with the computational complexity of an AI algorithm, often at the cost of its completeness. Moreover, a heuristic is generally specialized: its use gives good performance on particular types of problems, but is not able to solve others. We think that using several diversified heuristics rather than a single one may deal with these difficulties. Two possibilities can be considered:

- The heuristics are executed competitively for each problem, and the system chooses to execute the solution given by the first heuristic to satisfy an evaluation criterion, for example the first heuristic to find a solution, or the first to find a *correct* solution in spite of development faults if this correctness may be assessed by a corresponding acceptance test, e.g., by plan analysis as previously presented (compensation recovery).
- One heuristic is privileged and executed first; other heuristics may be successively tried if no (correct) solution is found in a set time (forward recovery).

Either way, a set of heuristics adapted to the different problems that the decisional mechanism may face needs to be identified. An analysis and classification of the heuristics regarding the type of problems that they solve efficiently may be particularly useful to help choose the successive order of the heuristics to be executed in our forward recovery proposition.

4) Model diversification: Diversification of the model consists in the implementation of different descriptions of the system, environment, tasks, or procedures used by the decisional mechanism to draw inferences. The decisional mechanism detects the presence of an adverse situation when an executive procedure or a plan produced from one of these models fails (although it is unable to ascertain whether this situation is caused by a system fault or by an environmental uncertainty). It may then try to recover from the possible failure by switching to another model and re-planning. This technique aims to tolerate development faults in the model descriptions, in a similar way to software diversification in classic computing systems. Whether it is efficient for autonomous systems remains to be seen.

B. Fault tolerance comparison between different decisional mechanisms

The implementation of any fault tolerance mechanism often negatively impacts the system performance and development costs, principally because it requires redundancy at some level to detect an error or accomplish a recovery. Evaluation of such mechanisms is thus essential to ascertain whether their effectiveness outweighs their overheads. We present in this section the simulation and fault injection environment that we are currently implementing to experimentally evaluate the efficiency of the various possibilities previously presented.

The use of simulation rather than the execution of the actual system is mainly motivated by two reasons:

- The large number of experiments required to perform a significant evaluation: experiments on real systems usually need more time to be executed, and are more difficult to automate.
- The hazardous behavior of the system during an experiment: as we inject faults in the system, we cannot predict its behavior, which may cause damage to itself or its direct surroundings.

Previous studies have demonstrated that fault injection may efficiently simulate real software faults [8] [17]. We particularly focus on two techniques:

- Mutation of a program source is the introduction of a unique fault through the modification of a particular line of code. It simulates programming faults accurately, but is generally costly in time as one mutated version of the program must be compiled for each fault.
- Interception and modification of the inputs or outputs of a program aims to simulate errors caused by the fault rather than the fault itself.

1) Simulation environment: The simulation environment that we intend to use is represented in Figure 4. Different decisional mechanisms can be implemented on top of a hardware interface (composed of Genom modules from the LAAS architecture), providing requests to a virtual robot that operates as part of the simulation environment.

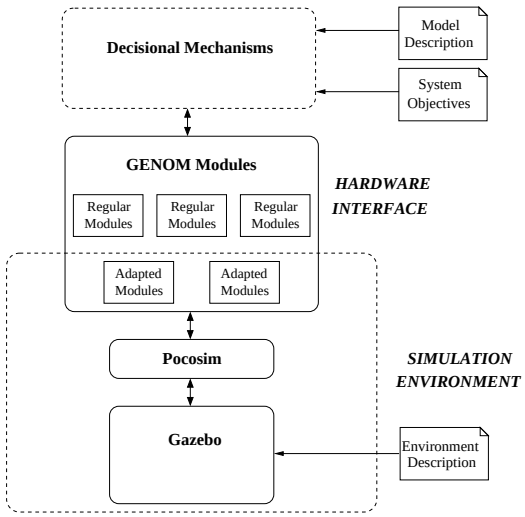

Figure 4: Simulation environment

This environment may be decomposed into three elements: a robot simulator named Gazebo [2], an interface library named Pocosim, and some modules from the functional layer of a three-layer type of architecture.

[2]From the "player/stage project", http://playerstage.sourceforge.net

- The robot simulator Gazebo is used to simulate the physical world and the actions of the autonomous system; it takes as input a file describing the environment of the simulation (mainly a list of static or dynamic obstacles containing their position, and the physical description of the robot) and executes the movement of the robot and dynamic obstacles, and possible interactions between objects.
- The Pocosim library [19] is a software bridge between the simulated robot (executed on Gazebo) and the software commands generated by the Genom modules: it transforms commands to the actuators into movements or actions to be executed on the simulated robots, and relays the sensor inputs that Gazebo produces from the simulation.
- The Genom modules [12] compose the hardware interface between decisional mechanisms and the simulated hardware: they execute non-decisional algorithms and generate commands to the simulated system from requests given by upper-level components. Special libraries must be used with the modules that are supposed to control the hardware components of the system, in order to link them with the Pocosim bridge; other modules are integrated as they are. Existing modules can be used without modifying their source code.

The Genom modules receive requests from, and send reports to, the upper decisional mechanisms, including the decisional mechanism to be evaluated. Typically, these decisional mechanisms may be a planner and an executive (as in a classic three-layer type of architecture), or a hierarchy of agents (as in multi-agent type of architecture).

Currently, we are integrating the different components of the simulation environment with the OpenPRS executive and the IxTeT-eXeC temporal executive and planner of the LAAS architecture. From this base, we should be able to implement the various possibilities evoked in section IV-A: most techniques imply only modifying (or possibly replacing) the IxTeT-eXeC component. However, the comparison with the agent-based architecture requires replacing both IxTeT-eXeC and OpenPRS by agents possessing *equivalent* knowledge in order for the comparison to be valid. This may prove difficult since the programming approaches for the two systems are not similar.

2) Fault injection: In order to establish a campaign of experiments for assessing the fault tolerance capabilities of a particular decisional mechanism, we need to determine four different aspects (Figure 5) [2]: the *faults* (or faultload) to be injected into the system, the *activity* (or workload) that the robot will execute during an experiment, the *results* that we need to observe and store, and the *measures* that we will use to sum up the results and compare the different systems. For the measures to be representative, a sufficient number of experiments must be performed: automation of experiments is thus imperative (e.g., through the use of shell scripts).

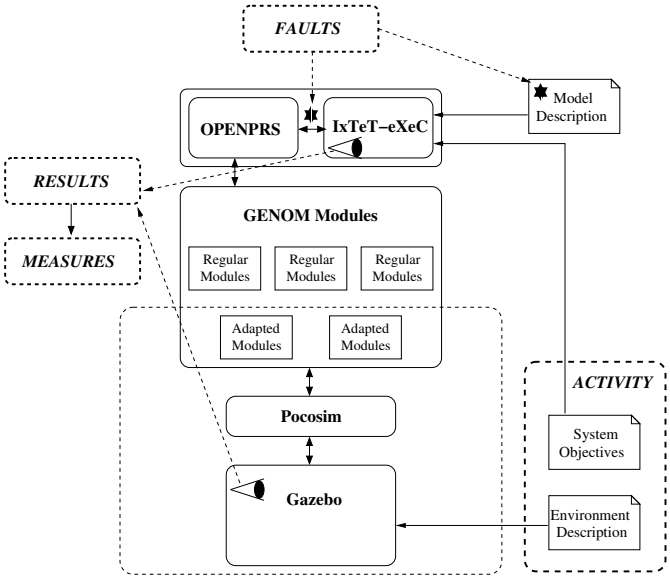

Figure 5: Proposed experimental framework

- *Faultload*: Each experiment focusses on a specific fault; to narrow our focus, we consider only *software faults affecting decisional mechanisms*. Faults may be injected either by mutating the model given to the planner, or by intercepting and modifying (for example by bit-flipping) a report or a request between the planner and the executive. As the model is generally directly parsed by the decisional mechanisms, there is no need to compile the mutants obtained from the model.
- *Workload*: The activity encompasses the task that the system must carry out, and the environment in which it will evolve. We have chosen the case of a space rover, required to take pictures of particular locations, to upload them into a remote database during given communication windows, and to return home in a set time. An environment is a set of static obstacles unknown to the robot (possibly blocking the system from executing one of its goals), which introduces uncertainties and stresses the system navigation mechanism. Each experiment should be composed of different runs using different environments, in order to activate the system's functionalities as broadly as possible.
- *Results*: To establish relevant measures, we need to gather results concerning whether the injected fault has been activated during the experiment, and possibly whether it has been detected and correctly identified by the system. We also need to know the list of different objectives that the system has successfully completed. Other useful results include the distance covered by the system and the duration of its activity. The information pertaining to system activity (completion of objectives, distance and duration) must be obtained from Gazebo during the execution of an experiment, whereas information concerning treatment of the fault and its subsequent error(s) must be obtained through observation of the targeted decisional mechanism.

- *Measures*: Measures consist of dependability-specific measures and other performance measures. Dependability measures encompass principally the *coverage* of each decisional mechanism implemented with respect to the faults injected and activated, and the error detection *latency*. Performance measures quantify the degree of success of the system regarding its mission: the mean number of goals it has achieved, and the mean time and distance required.

CONCLUSION

In this article, we have presented several basic concepts:

- Robustness and fault tolerance, both characterizing the resilience of a system towards particular adverse situations; *robustness* characterizes resilience towards uncertainties of the environment, and *fault tolerance* characterizes resilience towards faults affecting the system resources.
- *Decisional mechanisms* are central to autonomous systems as they embody the ability to dynamically select appropriate actions to achieve specific objectives; they are composed of *knowledge* specific to a domain of application and an *inference mechanism* used to solve problems.

We also summarized the state of the art in dependability and robustness mechanisms used in autonomous systems, and some conclusions that we drew from it. In particular:

- Fault avoidance is largely privileged compared to other dependability means, although it rarely appears to be implemented intensively enough.
- Development faults are hardly addressed by fault tolerant mechanisms in autonomous systems; robustness techniques somewhat compensate this problem, but are surely insufficient for critical applications.

We presented our specific area of interest: the development of fault tolerance mechanisms for autonomous systems, and more particularly techniques for tolerating residual faults in the knowledge used by decisional mechanisms. Two aspects can be considered:

- The *safety* aspect aims to minimize the probability of a catastrophic failure. We believe that the use of safety-bags may be efficient to detect unsafe system commands and react by positioning the system in a safe state, but more work is needed on the expression of safety constraints to be checked by the safety-bag.
- The *reliability* aspect distinguishes two cases: the acceptability of decisions and their timeliness. Reliability in the presence of faults requires error detection and system recovery. We are currently focussing on this particular aspect.

We finally described the current and future directions of our work. We presented four possible research paths to improve fault tolerance in decisional mechanisms:

- The use of *agent-based type of architecture* to take advantage of their flexibility and internal semantic consistency.
- *Plan analysis* for detecting errors on-line by checking the validity of plans before their execution.
- *Heuristics diversification* to tolerate residual faults in the heuristics or their inability to treat specific situations.
- *Model diversification* to implement redundant models to compensate for possible faults in domain-specific knowledge.

We also presented an experimental framework based on simulation and fault injection that we are currently implementing to assess the efficiency of these mechanisms.

REFERENCES

[1] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand. An Architecture for Autonomy. *The International Journal of Robotics Research*, 17(4):315–337, April 1998.

[2] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J. C. Fabre, J. C. Laprie, E. Martins, and D. Powell. Fault Injection for Dependability Validation - A Methodology and Some Applications. *IEEE Transactions on Software Engineering*, 16(2):166–182, 1990.

[3] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Trans. on Dependable and Secure Computing*, 1(1):11–33, January-March 2005.

[4] D. E. Bernard, E. B. Gamble, N. F. Rouquette, B. Smith, Y. W. Tung, N. Muscettola, G. A. Dorias, B. Kanefsky, J. Kurien, W. Millar, P. Nayal, K. Rajan, and W. Taylor. Remote Agent Experiment DS1 Technology Validation Report. Ames Research Center and JPL, 2000.

[5] M. C. Boden. Benefits and Risks of Knowledge-Based Systems, 1989. (ISBN 0-19854-743-9).

[6] J. J. Borrelly, E. Coste-Manière, B. Espiau, K. Kapellos, R. Pissard-Gibollet, D. Simon, and N. Turro. The ORCCAD Architecture. *The International Journal of Robotics Research*, 17(4):338–359, April 1998.

[7] J. Carlson and R. R. Murphy. Reliability Analysis of Mobile Robots. In *Proceedings of the 2003 IEEE International Conference on Robotics & Automation*, pages 274–281, Taipei, Taiwan, September 14-19, 2003.

[8] M. Daran. *Modelling Erroneous Software Behavior and Application to Testset Validation by Fault Injection*. PhD thesis, Institut National Polytechnique de Toulouse, 1996. LAAS Report No. 96497 (in French).

[9] O. Despouys. *An Integrated Architecture for Planning and Execution Control in Dynamic Environments*. PhD thesis, Institut National Polytechnique de Toulouse, 2000. LAAS Report No. 00541 (in French).

[10] C. Dousson. *Evolution Tracking and Chronical Recognition*. PhD thesis, Université Paul Sabatier, 1994. LAAS Report No. 94394 (in French).

[11] M. S. Feather and B. Smith. Automatic Generation of Test Oracles - From Pilot Studies to Application. In *Proceedings of the 14th IEEE Automated Software Engineering Conference (ASE-99)*, Cocoa Beach, Florida, October 12-15, 1999.

[12] S. Fleury, M. Herrb, and R. Chatila. Genom: A Tool for the Specification and the Implementation of Operating Modules in a Distributed Robot Architecture. In *Proceedings of the 1997 International Conference on Intelligent Robots and Systems (IROS 97)*, Grenoble, France, September 1997.

[13] J. Fox and S. Das. *Safe and Sound, Artificial Intelligence in Hazardous Applications*. The American Association for Artificial Intelligence Press, 2000. (ISBN 0-262-06211-9).

[14] E. Gat. On Three-Layer Architectures. In *Artificial Intelligence and Mobile Robots*, D. Kortenkamp, R. P. Bonnasso, and R. Murphy editors, MIT/AAAI Press, pages 195-210, 1997.

[15] R. P. Goldman, D. J. Musliner, and M. J. Pelican. Using Model Checking to Plan Hard Real-Time Controllers. In *Proceeding of the AIPS2k Workshop on Model-Theoretic Approaches to Planning*, Breckenridge, Colorado, April 14, 2000.

[16] B. Graf, M. Hans, and R. D. Schraft. Mobile Robot Assistants - Issues for Dependable Operation in Direct Cooperation With Humans. *IEEE Magazine on Robotics & Automation*, 11(2):67–77, 2004.

[17] M. T. Jarboui. *Computer System Dependability. Benchmarking and Fault Representativeness*. PhD thesis, Institut National Polytechnique de Toulouse, 2003. LAAS Report No. 03245 (in French).

[18] C. Jones, M. O. Killijian, H. Kopetz, E. Marsden, N. Moffat, M. Paulitsch, D. Powell, B. Randell, A. Romanovsky, and R. Stroud. Revised Version of DSoS Conceptual Model (DC1). Technical Report LAAS-CNRS No. 01441, EU IST-1999-11585 DSoS (Dependable Systems of Systems), 2001.

[19] S. Joyeux, A. Lampe, R. Alami, and S. Lacroix. Simulation in the LAAS Architecture. In *Proceedings of Principles and Practice of Software Development in Robotics (SDIR2005), ICRA workshop*, Barcelona, Spain, April 18, 2005.

[20] P. Klein. The Safety-Bag Expert System in the Electronic Railway Interlocking System Elektra. *Expert Systems with Apllications*, 3(4):499–506, 1991.

[21] N. Lecubin, J. C. Poncet, D. Powell, and P. Thévenod. SPAAS: Software Product Assurance for Autonomy on-board Spacecraft. Technical Report 01267, LAAS-CNRS, 2001.

[22] S. Lemai and F. Ingrand. Interleaving Temporal Planning and Execution in Robotics Domains. In *Proceedings of AAAI-04*, pages 617–622, San Jose, California, July 25-29, 2004.

[23] B. Lussier, R. Chatila, J. Guiochet, F. Ingrand, M. O. Killijian, and D. Powell. State of the Art on Dependability of Autonomous Systems. Technical Report 05226, LAAS-CNRS, 2005. (in French).

[24] B. Lussier, R. Chatila, F. Ingrand, M. O. Killijian, and D. Powell. On Fault Tolerance and Robustness in Autonomous Systems. In *Proceedings of the third IARP/IEEE-RAS/EURON Joint Workshop on Technical Challenge for Dependable Robots in Human Environments*, Manchester, GB, September 7-9, 2004.

[25] E. Marchand, E. Rutten, H. Marchand, and F. Chaumette. Specifying and Verifying Active Vision-Based Robotic Systems with the SIGNAL Environment. *The International Journal of Robotics Research*, 17(4):418–432, April 1998.

[26] Benoit Morisset, Guillaume Infantes, Malik Ghallab, and Felix Ingrand. Robel: Synthesizing and Controlling Complex Robust Robot Behaviors. In *4th International Cognitive Robotics Workshop*, Valencia, Spain, August 23-24, 2004.

[27] N. Muscettola, G. A. Dorais, C. Fry, R. Levinson, and C. Plaunt. IDEA: Planning at the Core of Autonomous Reactive Agents. In *AIPS 2002 Workshop on On-line Planning and Scheduling*, Toulouse, France, April 22, 2002.

[28] N. Muscettola, P. P. Nayak, B. Pell, and B. C. Williams. Remote Agent: To Boldly Go Where No AI System Has Gone Before. *Artificial Intelligence*, 103(1-2):5–47, 1998.

[29] N. J. Nilsson. *Artificial Intelligence: A New Synthesis*. Morgan Kaufmann Publishers, 1998. (ISBN 1-55860-467-7).

[30] F. Py and F. Ingrand. Real-Time Execution Control for Autonomous Systems. In *Proceedings of the 2nd European Congress ERTS, Embedded Real Time Software*, Toulouse, France, January 21-23, 2004.

[31] A. Ranganathan and S. Koenig. A Reactive Robot Architecture with Planning on Demand. In *Proceedings of the 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1462–1468, Las Vegas, Nevada, October 27-31, 2003.

[32] S. A. Schneider, V. W. Chen, G. Pardo-Castellote, and H. H. Wang. ControlShell: A Software Architecture for Complex Electromechanical Systems. *The International Journal of Robotics Research*, 17(4):360–380, April 1998.

[33] N. Tomatis, G. Terrien, R. Piguet, D. Burnier, S. Bouabdallah, K. O. Arras, and R. Siegwart. Designing a Secure and Robust Mobile Interacting Robot for the Long Term. In *Proceedings of the 2003 IEEE International Conference on Robotics & Automation*, pages 4246–4251, Taipei, Taiwan, September 14-19, 2003.

[34] R. Volpe, I. Nesnas, T. Estlin, D. Mutz, R. Petras, and H. Das. CLARAty: Coupled Layer Architecture for Robotic Autonomy. Technical Report D-19975, NASA - Jet Propulsion Laboratory, 2000.