Fault Tolerant Planning for Critical Robots

Benjamin Lussier, Matthieu Gallien, Jérémie Guiochet,
Félix Ingrand, Marc-Olivier Killijian, David Powell
LAAS-CNRS, 7 avenue du Colonel Roche, 31077 Toulouse Cedex 4, France
firstname.lastname@laas.fr

Abstract

Autonomous robots offer alluring perspectives in numerous application domains: space rovers, satellites, medical assistants, tour guides, etc. However, a severe lack of trust in their dependability greatly reduces their possible usage. In particular, autonomous systems make extensive use of decisional mechanisms that are able to take complex and adaptative decisions, but are very hard to validate. This paper proposes a fault tolerance approach for decisional planning components, which are almost mandatory in complex autonomous systems. The proposed mechanisms focus on development faults in planning models and heuristics, through the use of diversification. The paper presents an implementation of these mechanisms on an existing autonomous robot architecture, and evaluates their impact on performance and reliability through the use of fault injection.

1. Introduction

Autonomous systems cover a large range of functionalities and complexities, from robotic pets to space rovers, including elderly care assistants, museum tour guides, and autonomous vehicles. As successes arise in autonomous navigation, exemplified by Mars rovers and the clearing of the *DARPA Grand Challenge* [17], complex autonomous systems that are able to choose and execute high-level actions without human supervision are not yet ready for real life applications. Indeed, one of the major drawbacks in the utilization of such systems is the difficulty to predict and validate their behavior. To increase the confidence that we may have in such systems so that they may be used in more critical applications, we consider in this paper the tolerance of residual development faults in planning models.

First, we introduce autonomous systems and specific aspects such as decisional mechanisms, robustness and planning. Second, we propose error detection and recovery mechanisms that are appropriate for planning to tolerate development faults in their application-dependent knowledge. Finally, we validate the proposed mechanisms through an experimental framework based on fault injection.

2. Dependability in Autonomous Systems

This section presents several aspects of autonomous systems relevant to their dependability. We present a definition of autonomy and give key aspects of architectures for autonomous robots.

2.1. Autonomy

A dictionary definition of "autonomy" is "the ability to act independently". However, in the field of robotics, this definition is insufficient since it does not enable a distinction between classic *automatic* systems, that simply apply preprogrammed reactions in response to the system's inputs (e.g., as in feedback control), and truly *autonomous* systems that seek to carry out goal-oriented tasks whose implementation details are not defined in advance, either by necessity (the input space is unbounded) or as a design strategy (to simplify the code). We adopt here the definition of autonomy given in [11]: *"An unmanned system's own ability of sensing, perceiving, analyzing, communicating, planning, decision-making, and acting, to achieve its goals as assigned by its human operator(s) through designed human-robot interaction (HRI). Autonomy is characterized into levels by factors including mission complexity, environmental difficulty, and level of HRI to accomplish the missions."*

The level of autonomy of an autonomous system is often discussed in terms of its "robustness". Indeed, autonomous robots are intended to cope with uncertainty and non-nominal situations. A good or "robust" robot is understood to be one that can survive and fulfill its mission despite partial knowledge about its environment as well as unforeseen contingencies such as obstacles, rough terrain and failures. In this paper, we choose to distinguish between robustness and fault tolerance as follows:

Robustness is the ability of an autonomous system to

37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)
0-7695-2855-4/07 $20.00 © 2007 IEEE

cope with adverse environmental situations (lighting conditions, unexpected obstacles, etc.) while providing an acceptable service.

Fault Tolerance is the ability of an autonomous system to provide an acceptable service despite system faults (hardware failures, software bugs, etc.).

2.2. Autonomous System Architectures

Four architectural styles for designing autonomous robot systems are usually distinguished:

1. The *sense-plan-act style* is based on a closed loop of three components devoted respectively to sensing the environment, finding a plan to reach a goal state, and acting on the environment according to the plan.
2. The *subsumption style* allows several "behavior" components to simultaneously sense and act on the environment, with actions that can be prioritized or cross-inhibited between different components.
3. The *multi-agent style* considers a set of autonomous systems or agents immersed in the same environment and interacting to achieve their individual or shared goals.
4. The *hierarchical style* defines several abstraction levels with different real-time constraints, resulting in a layered architecture.

Whereas the sense-plan-act style has largely been abandoned (at least as the basis for a monolithic architecture) due to its poor real-time performance, the subsumption style is still commonly used in entertainment robots, such as Sony's *Aibo*™. The multi-agent style is now receiving considerable attention both as the basis for designing a taskable robot [18] and in the context of agent swarms with emerging "intelligence" [20]. However, most practical robots currently adopt the hierarchical style, usually resulting in an architecture with three layers [8]: (a) a *decisional layer* that is responsible for elaborating plans to reach operator-defined mission goals, (b) an *executive layer* that selects and sequences elementary actions that implement the high-level tasks included with the current plan, and (c) a *functional* layer that interfaces with the hardware sensory and action devices. In some architectures the executive layer is merged into either the decisional layer or the functional layer.

Hierarchical architectures for autonomy include the RAX architecture developed by NASA as part of its Deep Space One project [19], JPL's CLARATy [23] and the LAAS architecture [1] developed at LAAS-CNRS (the latter architecture will be described in more detail in 3.2.1). From a dependability viewpoint, tolerance of hardware faults is considered in some of these architectures [16]. For example, the RAX architecture includes a model-based *mode identification and reconfiguration* (MIR) component specifically aimed at diagnosing and recovering from faults affecting hardware resources [19]. For development faults, apart from on-line checking mechanisms aimed at guaranteeing safety [21], the focus has largely been on fault avoidance approaches (rigorous design, and thorough verification and testing). For example, intensive testing was carried out on the RAX architecture [3]: six test beds were implemented throughout the development process, incorporating 600 tests. The authors of [3] underline the relevance of intensive testing, but acknowledge particular difficulties regarding autonomous systems, notably the problem of defining suitable test oracles. Given the inherent difficulty of testing autonomous systems, we believe that a tolerance approach with respect to residual development faults should be of considerable interest. Yet, to the best of our knowledge, such an approach has not been previously envisaged.

2.3. Deliberation and Decision

From our perspective, deliberation and decision are the key features of autonomy. Many different decisional capabilities have been studied and deployed on robots or other autonomous systems. Here, we discuss what distinguishes such decisional capabilities from other programmed functionalities.

Most decisional mechanisms boil down to some sort of search in a very large state space. In general, this search leads to a decision (a plan to reach a goal, a diagnosis, an action, etc.). This search can be done either off-line or on-line, that is in advance to produce a precompiled data structure or on the fly while the system is running. It may reason about past states (as in diagnosis) or about future states (as in planning). It may have a limited horizon or, conversely, a very deep scope. But a key aspect is that the search needs to be efficiently guided to avoid a combinatorial explosion. As a result, a decisional mechanism can be *complete* (it is guaranteed to find a solution if one exists) or not (it can "miss it"), *correct* (solutions are always valid) or not (they are approximate), *tractable* (solutions are found in a polynomial time and space) or not.

Another important feature of decisional mechanisms is that they are organized in a way that makes a clear separation between the *knowledge* and the *inference mechanism*. The aim is to make the inference mechanism (e.g., a search engine) as generic and as independent as possible from the application. Conversely, the knowledge is domain-specific and typically specifies what states are reachable through the search process and what is the "best" next state from any given state. However, knowledge and inference mechanisms are often tightly linked in practice (e.g., heuristics that guide a search engine).

The implementation of decisional mechanisms relies on various formalisms (logic, neural networks, Markov models, constraints, simple temporal networks, etc.) and computational models (constraint-based programming, logic programming, heuristic search, dynamic programming, etc.).

The most common decisional functionalities deployed on autonomous systems are the following: planning, execution control, diagnosis, situation recognition and learning. In this paper, we focus particularly on *planning*, which is the activity of producing a plan to reach a goal from a given state, using given action models (e.g., the activity plan for the day of an exploration rover).

2.4. Planning

Planning is necessary in complex autonomous systems as a mean to select and organize the robot's future actions to achieve specified high-level goals. We introduce here some generalities on planning in autonomous systems, before presenting dependability issues.

2.4.1. General Principle. Planning can be implemented in several ways but, in practice, two approaches are preferred: search in a state space and constraint planning.

Search in a state space manipulates a graph of actions and states. It explores different action sequences from an initial state to choose the most suitable one to achieve given goals.

Constraint planning uses CSP (Constraint Satisfaction Problem) solving to determine a possible evolution of the system state that satisfies a set of constraints, some of which specify the system goals. CSP solving is commonly an iterative algorithm assigning successively possible values to each variable and verifying that all constraints remain satisfied.

Two robustness techniques are commonly implemented to recover from a plan failure caused by adverse environmental situations:

- *Replanning* consists in developing a new plan from the current system state and still unresolved goals. Depending on the planning model complexity, replanning may be significantly time costly. Other system activities are thus generally halted during replanning.
- *Plan repair* may be attempted before replanning, with the aim of reducing the time lost in replanning. It uses salvageable parts of the previous failed plan, that are executed while the rest of the plan is being repaired. However, if reducing the salvaged plan conflicts with unresolved goals, plan repair is stopped and replanning is initiated.

2.4.2. Dependability Issues. Planning, like other decisional mechanisms, poses significant challenges for validation. Classic problems faced by testing and verification are exacerbated. First, *execution contexts* in autonomous systems are neither controllable nor completely known; even worse, consequences of the system actions are often uncertain. Second, planning mechanisms have to be *validated in the complete architecture*, as they aim to enhance functionalities of the lower levels through high level abstractions and actions. Integrated tests are thus necessary very early in the development cycle. Third, the *oracle problem*[1] is particularly difficult since (a) equally correct plans may be completely different and (b) an unforeseen adverse environmental situation may completely prevent some goals from being achieved, thus ineluctably degrading the system performance, however well it behaves (for example, cliffs, or some other feature of the local terrain, may make a position goal unreachable).

One way to address the latter issue is to define an oracle as a set of constraints that necessarily and sufficiently characterizes a correct plan: plans satisfying the constraints are deemed correct. Such a technique was used for thorough testing of the RAX planner during the NASA *Deep Space One* project [3], or in the VAL validation tool [10]. Extensive collaboration of application and planner experts is necessary to generate the correct set of constraints. A Failure Recovery Analysis tool is proposed in [9] to ease model corrections during development.

Automatic static analysis may also be used to ascertain properties on planning models, whereas manual static analysis requires domain experts to closely scrutinize models proposed by planning developers. For example, the development tool Planware [2] offers facilities for both types of analysis.

Some work has also been done on evaluating planning dependability. A measure for planner reliability is proposed in [5], which compares theoretical results to experimental ones, showing a necessary compromise between temporal failures (related to calculability of decisional mechanisms) and value failures (related to correctness of decisional mechanisms). Later work [4] proposes concurrent use of planners with diversified heuristics to answer this compromise: a first heuristic, quick but dirty, is used when a slower but more focussed heuristic fails to deliver a plan in time. To our knowledge, no other fault tolerance mechanisms have been proposed in this domain. We strongly believe, however, that such mechanisms are essential to provide more dependability in autonomous systems.

3. Fault Tolerant Planning

We investigate here how to tolerate design and implementation faults in planner models and heuristics. These mechanisms are particularly well adapted to hierarchical

[1] How to conclude on correctness of a program's outputs to selected test inputs?

autonomous systems with a centralized planner at the decisional layer.

3.1. Principles

Complementary to testing, diversity is the only known approach to improve trust in the behavior of a critical system regarding development faults (e.g., diversification is used in software components of the Airbus A320, and in hardware components of the Boeing B777). The general principle of the mechanisms that we propose is to execute sequentially or concurrently diversified variants of the planner, following similar approaches to recovery blocks [22] and distributed recovery blocks [13]. In particular, diversity is encouraged by forcing the use of different algorithms, variable domains and parameters in the models and heuristics of the variants.

3.1.1. Detection. Implementing error detection for decisional mechanisms in general, and planners in particular, is difficult [16]. There are often many different valid plans, which can be quite dissimilar. Therefore, error detection by comparison of redundantly-produced plans is not a viable option. Thus, we must implement error detection by independent means. Here, we propose four complementary error detection mechanisms: a *watchdog timer*, a *plan analyzer*, a *plan failure detector* and an *on-line goal checker*.

A watchdog timer is used to detect when the search process is too slow or when a critical failure such as a deadlock occurs. Timing errors can be due to faults in the planner model, in its search engine, or ineffectiveness of the search heuristics.

A plan analyzer can be applied on the output of the planner. It is an acceptance test (i.e., an on-line oracle) that verifies that the produced plan satisfies a number of constraints and properties. This set of constraints and properties can be obtained from the system specification and from domain expertise but it must be diverse from the planner model. This mechanism is able to detect errors due to faults in the planner model or heuristics, and in the planner itself.

A plan failure detector is a classical mechanism used in robotics for execution control. Failure of an action which is part of the plan may be due to an unresolvable adverse environmental situation, or may indicate errors in the plan due to faults in the knowledge or in the search engine. Usually, when such an action failure is raised, thesearch engine tries to repair the plan. When this is not possible, it raises a plan failure. We use these plan failure reports for detection purposes.

An on-line goal checker verifies whether goals are reached while the plan is executed. Goals can only be declared as failed when every action of the plan has been carried out. This implies that the checker maintains an internal representation of the system state and of the goals that have been reached.

```
1. begin mission
2.   failed_planners ← ∅;
3.   while (goals ≠ ∅)
4.     candidates ← planners;
5.     while (candidates ≠ ∅ & goals ≠ ∅)
6.       choose k such as (k ∈ candidates)
                       & (k ∉ failed_planners);
7.       candidates ← candidates \ k;
8.       init_watchdog(max_duration);
9.       send (plan) to k;
10.      wait % we wait any of these two events
11.        □ receive (plan_found) from k
12.          stop watchdog;
13.          if analyze(plan)=OK then
14.             failed_planners ← ∅;
15.             k.execute_plan();
                % if the plan fails goals != empty
                % and then we loop line 3
16.          else
17.             send(invalid_plan) to operator;
18.             failed_planners ← failed_planners ∪ k;
19.          end if
20.        □ watchdog timeout
21.          failed_planners ← failed_planners ∪ k;
22.       end wait
23.       if failed_planners = planners then
24.         raise exception "no valid plan
                             found in time";
          % no remaining planner,
          % the mission has failed
25.       end if
26.     end while
27.   end while
28. end mission
```

Figure 1. Sequential Planning Policy

3.1.2. Recovery. We propose two recovery mechanisms, both using different planners based on diverse knowledge.

With the first mechanism, the planners are executed sequentially, one after another. The principle is given in Figure 1. Basically, each time an error is detected, we switch to another planner until all goals have been reached or until all planners fail in a row. Once all the planners have been used and there are still some unsatisfied goals, we go back to the initial set of planners. This algorithm illustrates the use of the four detection mechanisms presented in Section 3.1.1: watchdog timer (lines 8 and 20), plan analyzer (line 13), plan failure detector (line 15), on-line goal checker (lines 3 and 5).

Reusing planners that have been previously detected as failed makes sense for two different reasons: (a) a perfectly correct plan can fail during execution due to an adverse environmental situation, and (b) some planners, even faulty, can still be efficient for some settings since the situation that activated the fault may have disappeared.

It is worth noting that the choice of the planners, and the order in which they are used, is arbitrary in this particular example (line 6). However, the choice of the planner could take advantage of application-specific knowledge about the most appropriate planner for the current situation or knowledge about recently observed failure rates of the planners.

With the second recovery mechanism, the planners are executed concurrently [15]. The main differences with respect to the algorithm given in Figure 1 are: (a) the plan request message is sent to every planning candidate, (b) when a correct plan is found, the other planners are requested to stop planning, and (c) a watchdog timeout means that all the planners have failed.

Here, the choice of planner order is implicit: the first planner obtaining a plan is chosen. However, this could lead to the repeated selection of the same faulty but rapid planner. Some additional mechanism is thus required to circumvent this problem. For example, the planner selected during the previous round can be withdrawn from the set of candidates for the current round.

3.1.3. Coordination. From a dependability point of view, the fault-tolerance mechanisms have to be as independent as possible from the decisional layer, i.e., in this case from the planners. This is why we propose to handle both the detection and recovery mechanisms and the services necessary for their implementation in a middleware level component called FTplan, standing for *Fault-Tolerant PLANner coordinator*.

This component has to integrate the fault tolerance mechanisms into the robot architecture. This implies essentially communication between, and synchronization and coordination of, the error detection mechanisms and the redundant planners.

To avoid error propagation from a possible faulty planner, FTplan should not take any information that comes from or depends on the planners themselves. The watchdog can easily be implemented from the operating system timing primitives. Action failure detection is performed at the execution control layer, so error reports can be obtained and reused at the FTplan level. A plan analyzer performs simple acceptance checks using rules expressed independently from the planners and their knowledge.

However, implementing an on-line goal checker without relying on information obtained through the planner is more difficult. FTplan maintains for this purpose its own system state representation, based on information gathered from the lower layers. It obtains this information from the execution control layer, whose abstraction level is near to that of the decisional layer. This system state representation is checked against the set of goals prescribed for the current mission.

Whatever the particular recovery mechanism it implements, sequential or parallel, FTplan has to manage several planners. It needs to communicate with them, e.g., for sending plan requests or for updating their goals and system state representations before replanning. It also needs to be able to control their life cycle: start a new instance or even stop one when it takes too long to produce a plan.

FTplan is intended to allow tolerance of development faults in planners (and particularly in planning models). FTplan itself is *not* fault-tolerant, but being much simpler than the planner it coordinates, we can safely rely on classic verification and testing to assume that it is fault-free.

3.2. Implementation

We present here the implementation of the proposed mechanisms. We introduce the target architecture and then give some implementation details about the FTplan component.

3.2.1. LAAS architecture. The LAAS architecture is presented in [1], and some recent modifications have been proposed in [14]. It has been successfully applied to several mobile robots, some of which have performed missions in real situations (human interaction or exploration). It is composed of three main components[2] as presented in Figure 2: GenoM modules, OpenPRS, and IxTeT.

The functional level is composed of a set of automatically generated *GenoM modules*, each of them offering a set of services, which perform computation (e.g., trajectory movement calculation) or communication with physical devices (sensors and actuators).

The procedural executive *OpenPRS* (*Open Procedural Reasoning System*), is in charge of decomposing and refining plan actions into lower-level actions executable by functional components, and executing them. This component links the decisional component (IxTeT) and the functional level. During execution, OpenPRS reports any action failures to the planner, in order to re-plan or repair the plan. As several IxTeT actions can be performed concurrently, it has also to schedule sequences of refined actions.

IxTeT (*IndeXed TimE Table*) is a temporal constraint planner as presented in Section 2.4.1, combining high level actions to build plans. Each *action* is described in a *model* file used by the planner as a set of constraints on attributes (e.g., robot position), resources (e.g., energy consumption), numeric or temporal data (e.g., action duration). Then, a valid plan is calculated combining a set of actions in such a way that they are conflict-free and they fulfill the goals. The description of actions in the planner model is critical for the generation of successful plans and thus for the dependability of the robot as a whole.

3.2.2. Fault Tolerant Planner Implementation. The fault tolerance principles presented in Section 3.1 have been implemented in a fault tolerant planner component as presented in Figure 3. This component replaces the original component "Planner" presented in Figure 2. The FTplan component is in charge of communicating with OpenPRS

[2] An additional robustness component, R2C, is introduced in [21]. We have not considered it in this study since its current implementation is not compatible with our experimentation environment.

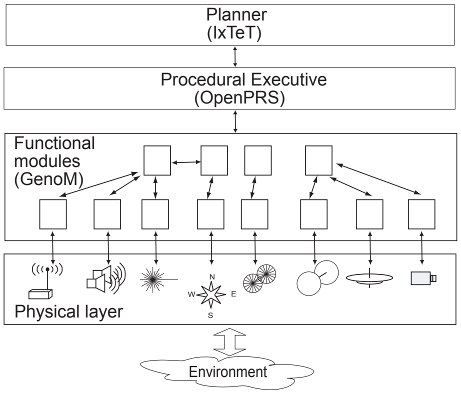

Figure 2. The LAAS architecture

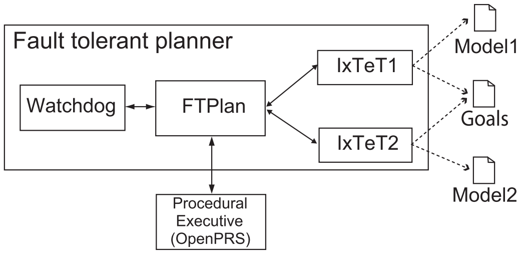

Figure 3. Fault tolerant planner

as the original planner does. To be consistent with the current implementation, FTplan uses the same technologies as OpenPRS and IxTeT for communication.

The current version of FTplan implements the sequential redundant planner coordination algorithm presented earlier (Section 3.1, Figure 1) with two IxTeT planners. Currently, the plan analysis function is empty (it always return *true*) so error detection relies solely on just three of the mechanisms presented in Section 3.1.1: watchdog timer, plan failure detection, and on-line goal checker.

The watchdog timer is launched at each start of planning. As soon as a plan is found before the time limit (40 seconds in our implementation: a sufficient time to produce plans in our activities), the watchdog is stopped. If timeout occurs, FTplan stops the current IxTeT, and sends a plan request to the other IxTeT planner, until a plan is found or both planners have failed. In the latter case, the system is put in a safe state (i.e., all activities are ceased), and an error message is sent to the operator.

On-line goal checker is performed after each action executed by OpenPRS that can result in a modification in the goal achievements (for instance: a camera shot, a communication, movement of the robot, etc.). This checking is carried out by analyzing the system state at the end of an action, determining goals that may have been accomplished and checking that no inconsistent actions have been executed simultaneously. Unfulfilled goals are resubmitted to the planner during the next replanning or at the end of plan execution.

In the actual implementation, FTplan checks every 10ms if there is a message from OpenPRS or one of the IxTeT planners. In case of an action request from a planner or an action report from OpenPRS, FTplan updates its system representation before transmitting the request. If the request is a plan execution failure (the system has not been able to perform the actions of the plan), then FTplan launches a re-plan using the sequential mechanism. If the request indicates that the actions are finished, then FTplan checks if the goals have been reached.

4. Mechanism Validation

We present here the validation process we have followed to assess the performance and efficacy of the proposed fault tolerant mechanisms. We discuss first a validation framework that extensively uses simulation and fault injection, then present experimental results.

4.1. Framework for Validation

Our validation framework relies on simulation and fault injection. Simulation is used since it is both safer and more practical to exercise the autonomy software on a simulated robot than on a real one. Fault injection is used since it is the *only* way to test the fault tolerance mechanisms with respect to their specific inputs, i.e., faults in planning knowledge. In the absence of any evidence regarding real faults, there is no other practical choice than to rely on *mutations*[3], which have been found to efficiently simulate real faults in imperative languages [7].

We now introduce successively the targeted software architecture, the workload, the faultload, and the readouts and measurements we obtain from system activity.

4.1.1. Software Architecture. Our simulation environment is represented in Figure 4. It incorporates three elements: an open source robot simulator named Gazebo, an interface library named Pocosim, and the components of the LAAS architecture already presented in section 3.2.1.

The *robot simulator Gazebo*[4] 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.

[3] A mutation is a syntactic modification of an existing program.

[4] "The player/stage project", http://playerstage.sourceforge.net

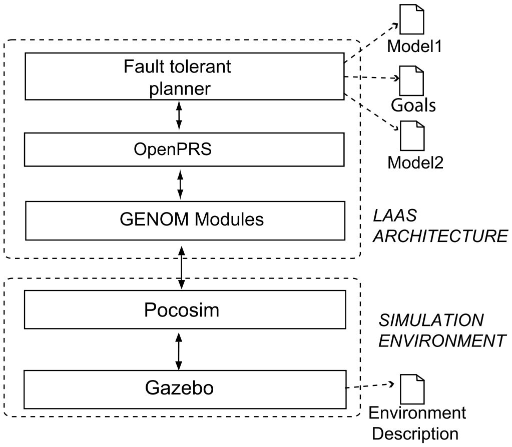

Figure 4. Simulation environment

The *Pocosim library* [12] 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 robot, and relays the sensor inputs that Gazebo produces from the simulation.

Our *autonomous system* is based on an existing ATRV (All Terrain Robotic Vehicle) robot, and employs GenoM software modules interfaced with the Gazebo simulated hardware. The upper layer of the LAAS architecture executes as presented in the previous section. Two different models are used with the IxTeT planners. The first model was thoroughly tested and used on a real ATRV robot; we use it as primary model and as target for fault injection. We specifically developed the second model through forced diversification of the first: for example, the robot position is characterized numerically in the first model and symbolically in the second.

4.1.2. Workload. Our workload mimics the possible activity of a space rover. The system is required to achieve three subsets of goals: *take science photos* at specific locations (in any order), *communicate* with an orbiter during specified visibility windows, and *be back at the initial position* at the end of the mission.

To partially address the fact that the robot must operate in an open unknown environment, we chose to activate the system's functionalities in some representative situations resulting from combinations of sets of missions and worlds. A *mission* encompasses the number and location of photos to be taken, and the number and occurrence of visibility windows. A *world* 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.

We implemented four missions and four worlds, thus applying sixteen execution contexts to each mutation. Missions are referenced as gradually more difficult M1, M2, M3 and M4: M1 consists in two communications and three photos in close locations, whereas M4 consists in four communications and five far apart photos. Environments are referenced as worlds W1, W2, W3 and W4. W1 is an empty world, with no obstacles to hinder plan execution. W2 and W3 contains small cylindrical obstacles, whereas W4 includes large rectangular obstacles that may pose great difficulties to the navigation module, and are susceptible to endlessly block the robot path.

In addition, several equivalent experiments are needed to address the non-determinacy of the experiments. This is due to asynchrony in the various subsystems of the robot and in the underlying operating systems: task scheduling differences between similar experiments may degrade into task failures and possibly unsatisfied goals, even in the absence of faults. We thus execute each basic experiment three times, leading to a total of 48 experiments per mutation. More repetition would of course be needed for statistical inference on the basic experiments but this would have led to a total number of experiments higher than that which could have been carried out with the ressources available (each basic experiment lasts about 20 minutes).

4.1.3. Faultload. To assess performance and efficacy of the proposed fault tolerance mechanisms, we inject faults in a planning model by random mutation of the model source code (i.e., in Model1 of Figure 3). Five types of possible mutations were identified from the model syntax:

1. Substitution of numerical values: each numerical value is exchanged with members of a set of real numbers that encompasses (a) all numerical variables in all the tasks of the model, (b) a set of specific values (such as 0, 1 or -1), and (c) a set of randomly-selected values.
2. Substitution of variables: since the scope of a variable is limited to the task where it is defined, numerical (resp. temporal) variables are exchanged with all numerical (resp. temporal) variables of the same task.
3. Substitution of attribute values: in the IxTeT formalism, attributes are the different variables that together describe the system state. Attribute values in the model are exchanged with other possible values in the range of the attribute.
4. Substitution of language operators: in addition to classic numerical operators on temporal and numerical values, the IxTeT formalism employs specific operators, such as "nonPreemptive" (that indicates that a task cannot be interrupted by the executive).
5. Removal of a constraint relation: a randomly selected constraint on attributes or variables is removed from the model.

Substitution mutations were automatically generated using the SESAME tool [6]. Using an off-line compilation, this tool detects and eliminates binary equivalent or syntactically incorrect mutants. Removal of random constraint relations was carried out through a PERL script and added to the mutations generated by SESAME. All in all, more than 1000 mutants were generated from the first model.

For better representativeness of injected faults, we consider only mutants that are able to find a plan in at least one mission (we consider that models that systematically fail would easily be detected during the development phase). As a simple optimization, given our limited resources, we also chose to carry out a simple manual analysis aimed at eliminating mutants that evidently could not respect the above criterion.

4.1.4. Records and Measurements. Numerous log files are generated by a single experiment: simulated data from Gazebo (including robot position and hardware module activity), output messages from GenoM modules and Open-PRS, requests and reports sent and received by each planner, as well as outputs of the planning process.

Problems arise however in trying to condense this amount of data into significant relevant measures. Contrary to more classic mutation experiments, the result of an experiment cannot be easily dichomotized as either failed or successful. As previously mentioned, an autonomous system is confronted with partially unknown environments and situations, and some of its goals may be difficult or even impossible to achieve in some contexts. Thus, assessment of the results of a mission must be graded into more than just two levels. Moreover, detection of equivalent mutants is complexified by the non-deterministic context of autonomous systems

To answer these issues to some extent, we chose to categorize the quality of the result of an experiment with: (a) the subset of goals that have been successfully achieved, and (b) performance results such as the mission execution time and the distance covered by the robot to achieve its goals. Due to space constraints, we focus in the rest of this paper on measurements relative to the mission goals.

4.2. Results

We present in this part several experimental results using the evaluation framework previously introduced. Experiments were executed on i386 systems with 3.2 GHz CPU and the Linux OS. We first study the performance cost of the proposed mechanisms, then present their efficacy in tolerating injected faults.

4.2.1. Fault-free Performance. To determine the overhead of the proposed fault tolerance mechanisms, we first concentrate on supposed fault-free models. Figure 5 presents the impact of FTplan on the system behavior.

Note that results in W4 must be treated with caution, as this world contains large obstacles that may cause navigational failures and block the robot path forever. As our work focus on planning model faults rather than limitations of functional modules, we consider that success in this world relies more on serendipity in the choice of plan rather than correctness of the planner model. It is however interesting to study the system reaction to unforeseen and unforgiving situations that possibly arise in an open and uncontrolled environment. Note that these results show that different models give rise to different failure behaviors: particularly in W4, the three systems fail differently.

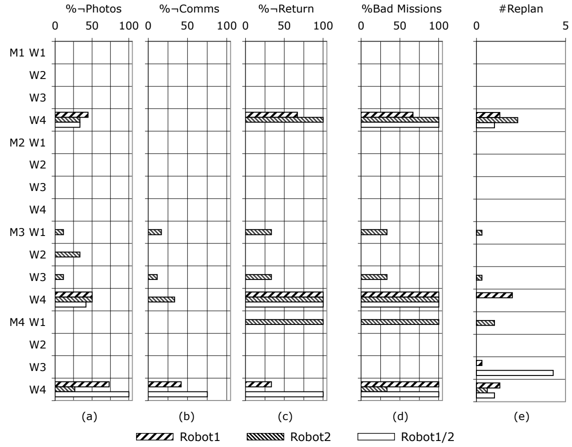

Figure 5. Impact of FTplan (without injected faults): *This figure studies the impact of the FTplan component on fault-free system behavior by comparing three different robots: Robot1 uses our first model, Robot2 uses our second model, and Robot1/2 contains an FTplan component that uses successively our first and second models. For each considered activity (M1W1 to M4W4), the figure shows five different measures: (a) (b) (c) three failure proportions to reach the different types of goals in a mission (resp. photos, communications, and returns to initial position), (d) failure proportion of the whole mission (a mission is considered failed if one or more mission goals were not achieved), and (e) the mean number of replanning operations observed during one experiment (in the case of Robot1/2, this number is equivalent to the number of model switches during the mission).*

W4 set aside, results are globally very good: Robot1 and Robot1/2 succeed in all their goals, while Robot2 fails a few goals in M3, and all its return goals in M4W1. These failures may be attributed to a larger set of constraints in this model that may be costly in performance, and underestimated distance declarations. The mean activity time of the systems (that is the time until the system stops all activity in a mission) is an average of 404 seconds for Robot1, 376 seconds for Robot2, and 405 seconds for Robot1/2. Time performance-wise, the three systems are thus roughly

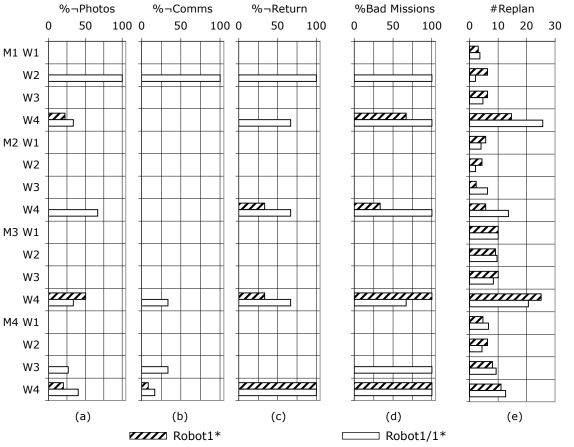

Figure 6. Impact of switching overhead (without injected faults): *This figure presents the impact of numerous model switches on fault-free system behavior by comparing two different robots: Robot1* uses our first model, and Robot1/1* contains an FTplan component that switches between two exact copies of our first model. Consequently, neither of the robots is fault-tolerant. The results concern* only *the cost of model switches. To provoke many model switches, both robots use a version of the IxTeT planner without the optimizing functionality of plan repair: any failed action systematically leads to complete replanning, with an additional model switch for Robot1/1*.*

equivalent.

Although the results are mostly positive, showing that FTplan's main execution loop does not severely decrease goal achievement or performance in the chosen scenarios, they are still insufficient to assess the overhead of planner switches as very few occurred in these fault-free experiments. This overhead is further studied in experiments presented in Figure 6.

We effectively see that there are many more replannings (and thus model switches) than in the previous experiments (a mean per experiment of 8.3 against 0.3 for Robot1*, and 8.9 against 0.4 for Robot1/1*). M1W2 appears as a singularity for Robot1/1* as after a few minutes of execution, the IxTeT planner finds no solution in its current situation. We believe this due to an elusive bug in either the model, FTplan, or the IxTeT planner. However, the same experiment with Robot1/2* (using diversification through the first and second models) gives successful missions, suggesting that the bug lies in our modified version of IxTeT.

Apart from this singularity, Robot1/1* only fails more goals than Robot1* in the over-stressing W4 execution contexts, as well as the complex M4W3. Setting aside W4 (and the M1W2 singularity), the mean activity time of the systems is 381 seconds for Robot1* and 431 seconds for Robot1/1*, indicating an overhead of 13%. Including W4, the time is 456 seconds for Robot1* and 535 seconds for Robot1/1*, indicating an overhead of 17%. We deem these results as quite acceptable considering the negative impact of discarding plan repair.

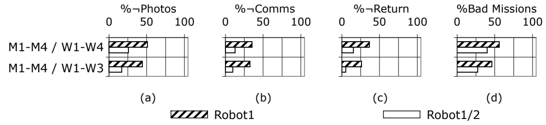

Figure 7. Impact of planner redundancy (with injected faults): *This figure presents overall results achieved for all 28 mutations, both with and without the heavily-constrained world W4.*

4.2.2. Fault-tolerance efficacy. To test the efficacy of the proposed mechanisms and the FTplan component, we injected 38 faults in our first model, realizing more than 3500 experiments equivalent to 1200 hours of testing. We discarded 10 mutants that were unable to find a plan for any of the four missions[5]. We believe that five of the remaining mutants are equivalent to the fault-free model. However, the non-deterministic nature of autonomous systems makes it delicate to define objective equivalence criteria. We thus include the results obtained with these five mutants.

The 28 considered mutations are categorized in the following manner: three substitutions of attribute values, six substitutions of variables, ten substitutions of numerical values, four substitutions of operators, and six removals of constraints. The mutants were executed on Robot1 and Robot1/2. The results are presented in Figure 7.

These results give objective evidence that model diversification favorably contributes to fault tolerance of an autonomous system considering the proposed faultload: failure decreases for photo goals of 62% (respectively, 50% including W4), 70% (64%) for communication goals, 80% (58%) for returns goals, and 41% (29%) for whole missions. Note, however, that RobotFT in the presence of injected faults is *less* successful than a single fault-free model. This apparent decrease in dependability is explained by the fact that incorrect plans are only detected when their execution has failed, possibly rendering one or more goals unachievable, despite recovery. This underlines the importance of plan analysis procedures to attempt to detect errors in plans *before* they are executed.

5. Conclusion

To our knowledge, the work presented in this paper is the first proposition of fault tolerant mechanisms based on di-

[5] In this case, Robot1/2 gives the same results as the fault-free Model2: nearly perfect success rates in W1, W2 and W3.

0-7695-2855-4/07 $20.00 © 2007 IEEE

versified planning models. We proposed a component providing error detection and recovery appropriate for fault-tolerant planning, and implemented it in the LAAS architecture. This component can use four detection mechanisms (watchdog timer, plan failure detector, on-line goal checker and plan analyzer), and two recovery policies (sequential planning and concurrent planning). Our current implementation is that of sequential planning associated with the first three error detection mechanisms.

To assess the performance overhead and the efficacy of the proposed mechanisms, we developed a validation framework that exercises the software on a simulated robot platform, and carried out what we believe to be the first ever mutation experiments on declarative models. These experiments were conclusive in showing that the proposed mechanisms do not severely degrade the system performance in the chosen scenarios, yet usefully improve the system behavior in the presence of model faults.

There are many directions for future research. First, implementation of a plan analyzer should allow much better goal success levels to be achieved in the presence of faults since it should increase error detection coverage and provide lower latency. Implementation of the concurrent planning policy and comparison with the sequential planning policy are also of interest. Moreover, we would like to evaluate diversification on planning heuristics rather than just models and investigate also the additional detection capabilities of recent additions to the LAAS architecture [21]. Finally, many more experiments are needed to improve the statistical relevance of the results. The use of a large computer grid would drastically improve the number of experiments that could be executed in reasonable time and eliminate the need for manual inspection to remove trivial mutants.

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] M. Becker and D. R. Smith. Model Validation in Planware. In *ICAPS 2005 Workshop on Verification and Validation of Model-Based Planning and Scheduling Systems*, Monterey, California, June 6-7, 2005.

[3] 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.

[4] I. R. Chen. On the Reliability of AI Planning Software in Real-Time Applications. *IEEE Transactions on Reliability*, 46(1):81–87, March 1997.

[5] I. R. Chen, F. B. Bastani, and T. W. Tsao. On the Reliability of AI Planning Software in Real-Time Applications. *IEEE Transactions on Knowledge and Data Engineering*, 7(1):14–25, February 1995.

[6] Y. Crouzet, H. Waeselynck, B. Lussier, and D. Powell. The SESAME experience: from assembly languages to declarative models. In *Proceedings of the 2nd Workshop on Mutation Analysis (Mutation'2006)*, Raleigh, NC, November 7, 2006.

[7] M. Daran and P. Thévenod-Fosse. Software error analysis: a real case study involving real faults and mutations. In *Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis*, San Diego, California, January 8-10, 1996.

[8] 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.

[9] A. E. Howe. Improving the Reliability of Artificial Intelligence Planning Systems by Analyzing their Failure Recovery. *IEEE Transactions on Knowledge and Data Engineering*, 7(1):14–25, February 1995.

[10] R. Howey, D. Long, and M. Fox. VAL: Automatic Plan Validation, Continuous Effects and Mixed Initiative Planning using PDDL. In *16th IEEE International Conference on Tools with Artificial Intelligence*, Boca Raton, Florida, November 15-17, 2004.

[11] H. M. Huang, editor. *Autonomy Levels for Unmanned Systems (ALFUS) Framework*. Number NIST Special Publication 1011. 2004.

[12] 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.

[13] K. H. Kim and H. O. Welch. Distributed Execution of Recovery Blocks: An Approach for Uniform Treatment of Hardware and Software Faults in Real-Time Applications. *IEEE Transactions on Computers*, C-38:626–636, 1989.

[14] 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.

[15] B. Lussier. *Fault Tolerance in Autonomous Systems*. PhD thesis, Institut National Polytechnique de Toulouse, 2007 (in French).

[16] B. Lussier, A. Lampe, R. Chatila, F. Ingrand, M. O. Killijian, and D. Powell. Fault Tolerance in Autonomous Systems: How and How Much? In *Proceedings of the 4th IARP/IEEE-RAS/EURON Joint Workshop on Technical Challenge for Dependable Robots in Human Environments*, Nagoya, Japan, June 16-18, 2005.

[17] M. Monterlo, S. Thrun, H. Dahlkamp, D. Stavens, and S. Strohband. Winning the DARPA Grand Challenge with an AI Robot. In *American Association of Artificial Intelligence 2006 (AAAI06)*, Boston, MA, July 17-20, 2006.

[18] 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. http://citeseer.nj.nec.com/593897.html.

[19] 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.

[20] J. L. Pearce, B. Powers, C. Hess, P. E. Rybski, S. A. Stoeter, and N. Papanikolopoulos. Using virtual pheromones and cameras for dispersing a team of multiple miniature robots. *Journal of Intelligent and Robotic Systems*, 45:307–21, 2006.

[21] 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.

[22] B. Randell. System Structure for Software Fault Tolerance. *IEEE Transactions on Software Engineering*, SE-1:220–232, 1975.

[23] 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.