Fault tolerant planning: towards dependable autonomous robots

Benjamin Lussier, Jérémie Guiochet, Félix Ingrand, Marc-Olivier Killijian, David Powell

Noname manuscript No.
(will be inserted by the editor)

Fault tolerant planning: towards dependable autonomous robots

Benjamin Lussier · Jeremie Guiochet · Felix Ingrand · Marc-Olivier Killijian · David Powell

Received: date / Accepted: date

Abstract Complex autonomous robots such as autonomous vehicles or robotic guides are critical systems because their failures could have catastrophic and costly consequences on themselves and their immediate environment, including users and bystanders. Moreover, verification and validation of these systems, that includes decisional software, is a difficult and complex task, requiring high expertise. In practice, despite recent advances in formal verification techniques and intensive testing for autonomous vehicles, it is still not possible to guarantee elimination of all residual development faults. Another way to enhance the confidence placed in such software, is to consider tolerance mechanisms with regards to these faults. This article proposes such an approach for temporal planners which are a major class of decisional software components in complex autonomous systems. The proposed fault tolerance mechanisms focus on residual development faults in planning models and heuristics. They use four complementary detection mechanisms to detect planning errors. Recovery from possible errors is achieved using redundant diversified planning models. We present an implementation of the proposed architecture on an existing autonomous robot software architecture. We also describe a validation framework used to evaluate the cost and efficacy of the fault tolerance mechanisms using real robot software on simulated robot hardware, and fault injection in the declarative planning models. In this framework, the proposed fault tolerant mechanisms are shown to greatly improve the system reliability with no significant impact on performance.

Keywords Fault Tolerant Planner · Fault injection · Plan execution · Autonomous Robots

B. Lussier
Université Technologique de Compiègne, Paris, France
E-mail: benjamin.lussier@utc.fr

J. Guiochet · F. Ingrand · M.-O. Killijian · D. Powell
LAAS-CNRS, Université de Toulouse, France E-mail: {name.firstname}@laas.fr

Mathematics Subject Classification (2000) 68
Categories (1)(2)

1 Introduction

Autonomous systems cover a large range of functionalities and complexities, from robotic pets to space rovers and satellites, including museum tour guides, autonomous vehicles and next generation of ambient intelligent systems for assisting people [20]. Despite successes in autonomous navigation, exemplified by Mars rovers and the clearing of the *DARPA Grand Challenge* [28] and *DARPA Urban Challenge* [34], and more recently by the Google and Delphi experiment cars, fully autonomous systems are not yet accepted for real-life applications. Such systems should be able to choose and execute high-level actions without any human supervision, in practice using planners as a central decisional mechanism. However, one of the major stumbling blocks is the difficulty of verifying and validating the behavior of such decisional software in an open, unstructured and dynamic environment. One way to increase the confidence that can be placed in planners despite imperfect verification and validation is to consider a tolerance approach with regard to residual development faults (such as design faults and programming faults). We investigate such an approach in this paper, focussing on faults in the planner's declarative planning models. To the best of our knowledge, very little work has been published on such an approach whereas we have shown in [22][23] that such mechanisms can really improve the level of confidence that can be placed in an autonomous system. Possible reasons for this limited use may result from the following points :

- fault tolerant mechanisms must be implemented through redundancy in a context with limited resources (space, power, etc.)
- fault tolerance mechanisms proposed in control engineering are efficient for dealing with sensor or actuator faults, but they generally do not consider faults related to decisional levels.

The main contribution of this paper is to present a fault tolerant architecture targeting software faults in planners, extendable to most decisional mechanisms, and to validate it through fault injection experiments.

The paper is structured as follows. Section 2 introduces basic concepts of dependability, robust planning, and current means for increase confidence in planers. In Section 3, we present a framework for developing planners that are able to tolerate development faults in their application-dependent knowledge, and an implementation example on an existing robot architecture. An experimental framework is proposed in Section 4 for evaluating the efficacy of the proposed fault-tolerance mechanisms. Section 5 presents our evaluation results and discusses the relevance of planning model diversification in our application. Finally, Section 6 concludes and suggests future research directions.

2 Background

2.1 Dependability concepts

Dependability is defined as [3] the ability to deliver service that can justifiably be trusted. Dependability encompasses many attributes, such as reliability. In this paper, we focus on *safety* and *availability*, respectively the absence of catastrophic consequences on the user and the environment, and the readiness for correct service. Dependability's threats are failures, error and faults. A service *failure* happens when delivered service deviates from correct service. An *error* is a deviation in the system's state. Errors can propagate though the system and may ultimately lead to a failure. Finally, a *fault* is the adjudged or hypothesized cause of an error. A fault is active when it causes an error, otherwise it is dormant. In this paper, we focus on development faults in planners, such as design faults (an incorrect planning model, improperly used heuristics, etc.) or programming faults (programming mistakes in the inference mechanism, faulty variable values, etc.).

To avoid service failures that are more frequent and more severe than is acceptable, dependability proposes four means:

- *fault prevention* : to prevent the occurrence or introduction of faults (techniques coming from system engineering and good practices)
- *fault removal* : to reduce the number and severity of faults (mainly validation and verification techniques)
- *fault forecasting* : to estimate the present number, the future incidence, and the likely consequences of faults (risk analysis methods)
- *fault tolerance* : to avoid service failures in the presence of faults (redundancy, error detections, etc.).

The proposed approach in this paper is based on the very well known fault-tolerant software method called Recovery Block [32]. It implements error detection and system recovery to avoid system failures. The Recovery Block pattern, as presented in [2], includes N *diverse*, independent, and functionally equivalent software modules called "versions" from the same initial specification. These diverse blocks are usually obtained using different programming languages, compilers, etc. and developed by different teams These blocks are classified into primary and $N-1$ secondary versions. The primary version (Block) is executed first and submitted to an error detection test. If an error is detected, a secondary alternate version (Recovery Block) is executed and tested. This last step is repeated until either one alternate passes is tested error-free, or all alternates are exhausted and an overall system failure is reported. Complementary to verification and testing, this technique is the only known approach to improve trust in the behavior of a critical system regarding residual development faults. For example, diversification is used in software components of the Airbus A320, and in hardware and software components of the Boeing B777.

2.2 Robust planning in robotics

On-line real-time planning is essential for any system that claims to be autonomous and able to fulfill its goals in an unpredictable open environment. Planning is the activity of producing a plan to reach a goal from a given state (e.g., the mission goals for the upcoming day of an exploration rover), using a given *planning model*. Planning can be implemented in several ways but, in practice, two approaches predominate:

- *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.
- *Search in plan space* manipulates a graph of incomplete plans. It starts with an empty plan containing the initial state and the final state (the planner's objectives), then considers ways to refine it by adding possible and useful actions until the search comes up with a complete plan that satisfies the planner goals. Unlike the search in a state space, actions are not added sequentially to the plan: the first action added to the empty plan may be the second to be executed. CSP (Constraint Satisfaction Problem) solving is an iterative algorithm commonly used in this approach, assigning successively possible values to each of the system variables and verifying that constraints between the variables remain satisfied.

In both cases, the planner typically consists of two parts: (a) a declarative *planning model* describing the system, the objects it can interact with, the system's possible actions and the associated constraints, and (b) a planning search engine that can reason on the planning model and produce a plan of actions enabling goals to be reached. A planner typically need two inputs: the current state of the system (position, sensors and actuators status, etc.) and its mission objectives (*goals*). The planning model is specific to the application, while the planning engine may be independent from the application. However planning model and search engine are often tightly linked by heuristics, that are included within the model to guide the search of the engine. Moreover, the planning model must be written in a way exploitable by the planning engine, and is usually not easily translated to another engine, nor easily understood by human developers or testers.

The robustness of a planner, that is its ability to achieve the system's assigned goals despite adverse situations (lighting conditions, unexpected obstacles, etc.), may be attained through either implicit or explicit handling of adverse situations [24].

Robustness through implicit handling of adverse situations is typically achieved by commitment strategy: planners seek to produce flexible plans that contain as much latitude and adaptability as possible [14, 29]. The plan produced is in fact a family of plans consistent with the constraints of the system. Inflexible plans, where all actions parameters are defined in advance, including the start and end dates of each action [10], need to rely heavily on adapta-

tion capabilities at lower system layers to tolerate small discrepancies in plan execution.

Robustness through explicit handling of adverse situations, i.e., when actions of the current plan fail, may be achieved through two strategies:

- *Re-planning*, which 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*, which attempts to reduce the time lost in re-planning by salvaging parts of the previous failed plan, and executing them while the rest of the plan is being repaired. However, if reducing the salvaged plan conflicts with unresolved goals, plan repair is stopped and re-planning is initiated.

Note however that such approaches do not cover residual faults in the planning engine nor in the planning models.

2.3 Dependable robot planners

Dependability in planners have been mostly achieved using fault removal (static verification and testing) in the domain of task planning in robotics. Nevertheless, as presented in [27], the classic issues faced by verification and testing in control systems, are exacerbated for autonomous systems:

- *execution contexts* in autonomous systems are neither controllable nor completely known; even worse, consequences of the system actions are often uncertain.
- integrated planning mechanisms have to be *validated in a complete architecture*, since 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.
- the *oracle problem*[1] is particularly difficult since (a) equally correct plans may be completely different, (b) non-deterministic action outcomes and temporal uncertainties can cause otherwise correct plans to sometimes fail when executed, and (c) unforeseen adverse environmental situations may completely prevent any plan from achieving all its goals (for example, cliffs, or some other feature of the local terrain, may make a position goal unreachable).

Despite those issues, some work concentrated efforts on the verification of the planner engine using static verifier [6], or a model checking [16] in the Deep-Space One spacecraft. But the trickiest part of planner software is most certainly the planning model itself, since it changes from one application to another and dictates *in fine* what plans are produced. Typically, a planning

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

model and the associated application-specific search heuristics are constructed and refined incrementally and empirically, by testing them against a graded set of challenges [15]. However, a small change to either can have surprisingly dramatic changes in the planner's behavior, both in terms of accuracy and performance.

One way to validate a planning model is to define an oracle as a set of constraints that 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 [4], and is supported by the VAL validation tool [17]. However, extensive collaboration of application and planner experts is often necessary to generate the correct set of constraints. Moreover, when the plans produced by the planner are checked against the same constraints as those included in the planning model, this approach only provides confidence about the planning engine and not the application-specific planning model. Some work [19,30] has attempted to validate application-specific models by means of model-checking, which usually implies a manual conversion of the model into the syntax accepted by the model checker. This requires an intimate knowledge of the model checker and it is thus usually carried externally by a formal method expert, rather than by the system designer. However, some recent research has studied how this model transformation can be automated [7].

To the best of our knowledge, even if fault tolerance is common in real-time control systems [36] and has been applied to robotic software components [35] and mechanical parts [33], not much work has been done on the tolerance of development faults in planners or other decisional mechanisms. Some methods to detect errors during plan execution are studied in robotics (like monitoring plan execution [5,31,26]), but they do not focus on faults in the planner itself. Even for global approaches like in [11,25], the error detection is not actually focusing on development faults in the planner.

One exception is [9], which proposes a measure for planner software reliability and compares theoretical results to experimental ones, showing a necessary compromise between temporal failures (related to tractability of decisional mechanisms) and value failures (related to correctness of decisional mechanisms). Later work [8] addresses this compromise through a fault-tolerance approach based on concurrent use of planners with diversified heuristics: a quick but dirty heuristic is used when a slower but more focused heuristic fails to deliver a plan in time. To our knowledge, no other fault tolerance mechanisms have been proposed in this domain. It is our opinion, however, that such mechanisms are a useful complement to verification and testing for planners embedded within critical autonomous systems.

3 An approach of fault tolerant planning: FTPlan

This section presents our approach using diversity to tolerate development faults in planning models and heuristics. We first introduce the general principles of our approach, before giving an implementation example.

As previously states, we focus in this paper on development faults in planners, such as design faults (an incorrect model, improperly used heuristics, etc.) or programming faults (programming mistakes in the inference mechanism, faulty variable values, etc.).

3.1 General principles

In this section, we propose a fault tolerant planner architecture based on the Recovery Block pattern introduced in Section 2.1. It uses only two diverse blocks, with an additional component, called FTPlan, in charge of detecting errors and performing the recovery. We first introduce this FTPlan component, then detail its error detection mechanisms. Finally, we propose two policies for system recovery: sequential and concurrent.

3.1.1 FTplan component

From a dependability point of view, the fault-tolerance mechanisms have to be as independent as possible 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 autonomous system architecture. This implies essentially communication, synchronization and coordination between the error detection mechanisms and the redundant planners.

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 planners it coordinates, classic verification and testing (such as formal method or exhaustive testing) can be applied to check that it is fault-free.

3.1.2 Error detection

Implementing error detection for decisional mechanisms in general, and planners in particular, is complex [24]. 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:

1. A *watchdog timer* can be used to detect when the search process is too slow or when a critical failure such as a deadlock occurs. Timing errors detected

in this way can be due to faults in the planning model, in its search engine, or ineffectiveness of the search heuristics.
2. A *plan analyzer* (i.e., an on-line plan oracle) can apply an acceptance test to the produced plan to check that it 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 independent with respect to the planning model. A plan analyzer is able to detect errors due to faults in the planning model or heuristics, and in the planner itself.
3. A *plan failure detector* is a classic 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 planning engine. Usually, when such an action failure is raised, the planning engine tries to repair the plan. When this is not possible, it raises a plan failure. We can use these plan failure reports for detection purposes.
4. An *on-line goal checker* verifies whether goals are reached while the plan is executed. A plan can be declared as (partially) failed if every action of the plan has been carried out but not all goals have been achieved. The on-line goal checker can resubmit unfulfilled goals to the planner at the start of the next replanning.

Note that both watchdog timer and plan analyzer detect errors during planning and thus before plan execution, while the plan failure detector and the on-line goal checker monitor the plan execution itself.

3.1.3 System recovery

We propose two recovery mechanisms, both using different planners based on diverse knowledge. The first one applies a sequential planning policy whereas the second one uses a parallel policy.

With the first mechanism, the planners are executed sequentially, one after another. The principle is given in Fig. 1. Basically, each time an error is detected, we switch to another planner until all goals have been reached or until all planners fail one after another when starting from the same initial system state. In the latter case, no models allow the planner to tackle the planning problem successfully: an exception must be raised to inform the operator of mission failure and to allow the system to be put into a safe state (line 29). When all planners have been used but some goals are still unsatisfied, we revert to the initial set of planners (while block: line 4 to 32). This algorithm illustrates the use of the four detection mechanisms presented in Section 3.1.2: watchdog timer (lines 9 and 25), plan analyzer (line 14), plan failure detector (line 16 and 18), on-line goal checker (lines 4, 6 and 17).

Until all goals have been achieved, the proposed algorithm reuses planners that have previously been detected as failed (line 5). This 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

```
1. begin mission
2.   exec_failure ← NULL;
3.   failed_planners ← ∅;
4.   while (attainable_goals ≠ ∅)
5.     candidates ← planners;
6.     while (candidates ≠ ∅ & attainable_goals ≠ ∅)
7.       choose k such as (k ∈ candidates)
                  & (k ∉ failed_planners)
                  & ( (k ≠ exec_failure)
                  | (k ∪ failed_planners = candidates));
8.       candidates ← candidates \ k;
9.       init_watchdog(max_duration);
10.      send(plan_request) to k;
11.      wait % for either of these two events
12.        □ receive (plan_found) from k
13.          stop_watchdog();
14.          if analyze(plan)=OK then
15.             failed_planners ← ∅;
16.             res_exec ← k.execute_plan();
17.             update(attainable_goals);
18.             if res_exec ≠ OK then
19.                exec_failure ← k;
20.             end if
                % if the plan fails, then
                % attainable_goals != empty and the
                % online goal checker will loop line 3 or 5
21.          else
22.             log(k.invalid_plan);
23.             failed_planners ← failed_planners ∪ k;
24.           end if
25.         □ watchdog timeout
26.           failed_planners ← failed_planners ∪ k;
27.      end wait
28.      if failed_planners = planners then
29.        raise exception 'no valid plan found in time';
           % no remaining planner,
           % the mission has failed
30.      end if
31.    end while
32.  end while
33. end mission
```

Fig. 1 Sequential Planning Policy

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 7): we only chose to exclude the last planner that led to an execution failure. 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, presented in Fig. 2, the planners are executed concurrently. After the concurrent planning, one of the produced plans is chosen (the first plan produced in the given algorithm), then validated by the plan analyzer. If no error is found, other plannings are suspended and the chosen plan is executed. If an error is detected by the plan analyzer (thus before plan execution), another plan is selected and analyzed, until either a

plan is found without detected errors, or there are no more candidate planners, or the watchdog timer limit activates. In the latter cases, as for the sequential planning, an exception must be raised to inform the operator of mission failure and to allow the system to be put into a safe state (lines 25 and 29). Although the system is supposed to be autonomous, if all decisional mechanisms are found failing, we can not rely on them any longer and the system must be stopped from executing his mission to avoid possible catastrophic failures.

The main differences with respect to the sequential planning policy are that: (a) the plan request message is sent to every planning candidate (line 6), (b) when a correct plan is found, the other planners are requested to stop planning (line 14), and (c) a watchdog timeout means that all the planners have failed (line 28). This second technique is more appropriate when the user believe that the some of the diversified planners may not be able to find a solution to the problem (which is NP-complete problem), while others could (for example thanks to diversified heuristics). Note that the computational overhead of executing multiple planners simultaneously my be mitigated by executing each planner on a different processor in multi-core computers.

In the given algorithm, 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. We have chosen in this example to withdraw the planner selected during the previous round from the set of candidates for the current round, as its plan has led the system to an execution failure (line 4). Note that as each planner use different planning model (and possibly planning engine), each model possibly more efficient in some situations than others, we have no guarantee that the same planner will be the first one every time.

3.2 Implementation on a real robot

We present in this section an implementation of the previously proposed sequential planning policy in the LAAS hierarchical software architecture for autonomous systems.

3.2.1 LAAS architecture

The LAAS architecture [1,21] 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 layers as presented in Fig. 3.

The functional layer 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). A service request gives rise to the execution of

```
1.  begin mission
2.    exec_failure ← NULL;
3.    while (attainable_goals ≠ ∅)
4.      candidates ← planners \ exec_failure;
5.      init_watchdog(max_duration);
6.      send(plan_request) to candidates;
7.      while (candidates ≠ ∅)
8.        wait % for either of these two events
9.        □ receive (plan) from k ∈  candidates
10.         candidates ← candidates \ k;
11.         pause_watchdog();
12.         if analyze(plan)=OK then
13.           stop_watchdog();
14.           send (cancel_planning) to candidates;
15.           candidates ← ∅;
16.           res_exec ← k.execute_plan();
17.           update(attainable_goals);
18.           if res_exec ≠ OK then
19.             exec_failure ← k;
20.           end if
              % if the plan fails, then
              % attainable_goals != empty and
              % the online goal checker will loop to line 3
21.         else
22.           resume_watchdog;
23.           log(k.invalid_plan);
24.           if (candidates = ∅)
25.              raise exception 'no valid plan found in time';
                 % no remaining planner,
                 % the mission has failed
26.            end if
27.          end if
28.       □ watchdog_timeout
29.          raise exception 'no valid plan found in time';
             % no remaining planner,
             % the mission has failed
30.       end wait
31.     end while
32.   end while
33. end mission
```

Fig. 2 Concurrent Planning Policy

an elementary action, the success or failure of which is reported to the requester, along with other action-specific information. Data exchange between modules is performed through the use of "posters", each of which is a shared memory space attached to a module, and readable by the others.

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 coordinating their execution. This component links the decisional component (IxTeT) and the functional layer. 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*) [14] is a temporal constraint planner, combining high-level actions to build plans. It uses CSP to search in plan space, as presented in Section 2.2. Its deliberations are based on piecewise constant functions called *attributes* that represent the evolution of the system state, of

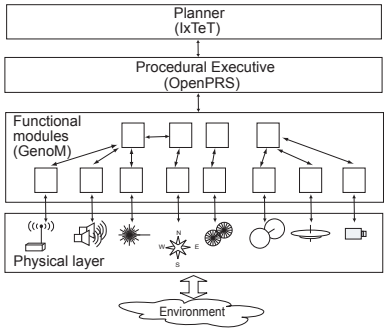

Fig. 3 The LAAS architecture

```
1.  task TAKE_PHOTO(?x, ?y)(t _start, t_end) {
2.    ?x in [-oo,+oo]; ?y in [-oo,+oo];
3.    hold(POS_X():?x, (t_start, t_end));
4.    hold(POS_Y():?y, (t_start, t_end));
5.    hold(POS _CAMERA():down,(t_start, t_end));
6.    event(IMAGE(?x, ?y):(to_do,done),t_end);
7.    use(CAMERA():1, (t_start, t_end));
8.    (t_start - t_end) in [10,60];
9.  }nonPreemptive
```

Fig. 4 An Action in the IxTeT Formalism

its resources, and of its environment. The different actions are described in a *planning model* file as a set of constraints either on the system attributes (e.g., robot position, energy consumption, and environment evolution) or on temporal and numerical variables (e.g., action duration). A valid plan is a partially-ordered set of possibly concurrent, non-conflicting actions that together achieve the system goals.

Constraints (C) are defined using:

- classical mathematical operators for temporal and numerical variables (V),
- the *consume*, *produce* and *use* predicates for consumption, production or usage of a system resource,
- the *hold* (H) and *event* (E) predicates for the other system attributes. The hold predicate represents persistence of an attribute value over a given period of time (e.g., *hold(robot, position, (start,end))*), whereas the event predicate represents an instantaneous change of value (e.g., *event(photo, (to_ do, done), time)*).

Actions are modeled through IxTeT *tasks*. Fig. 4 gives the example of a high-level action that can be used to photograph a scientific object in an exploratory mission. Line 1 declares the task and its numerical and temporal parameters: `x` and `y` are the Cartesian coordinates of the scientific object to be photographed, while `t_start` and `t_end` are temporal constraints representing respectively the starting and ending time of the task. Lines 3 to 5 define the

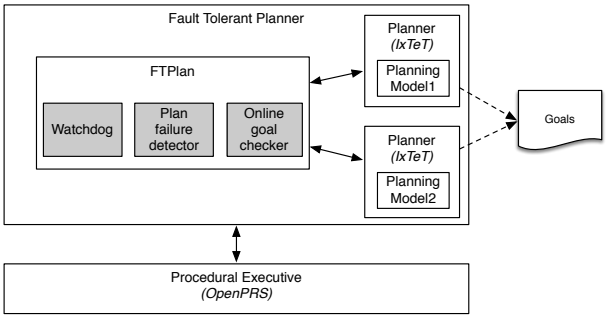

Fig. 5 Fault tolerant planner

constraints on the system attributes required for all the task duration: lines 3 and 4 stipulate that the robot position must not change while taking a picture, whereas line 5 requires that the camera points down to the object that needs to be photographed. Line 6 marks the photo as successfully taken when the task terminates. Line 7 presents an example of resource management: the resource CAMERA is used for the duration of the task. Line 8 presents an example of constraints on temporal values by specifying the possible duration of the task (from ten to sixty seconds). Finally, line 9 closes the task definition and states that executions of the task cannot be preempted.

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 Fig. 5. This component replaces the original component "Planner" presented in Fig. 3. The FTplan component is in charge of communicating with OpenPRS 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, Fig. 1) with two IxTeT planners. Currently, the plan analysis function is empty (it always returns *true*) so error detection relies solely on just three of the mechanisms introduced earlier: watchdog timer, plan failure detection, and on-line goal checker.

Fig. 6 presents an example of a fault tolerance scenario using the sequential policy: a first plan is produced and executed using the planner, but an action failure is detected during execution. To simplify the diagram, the plan failure detector service is represented with the message "executionFailure". The first planner is then re-initialized while the second one is asked for a new plan from the current situation. However this planning lasts too long (a model fault may have caused the planner to freeze) so the watchdog times out, FTplan

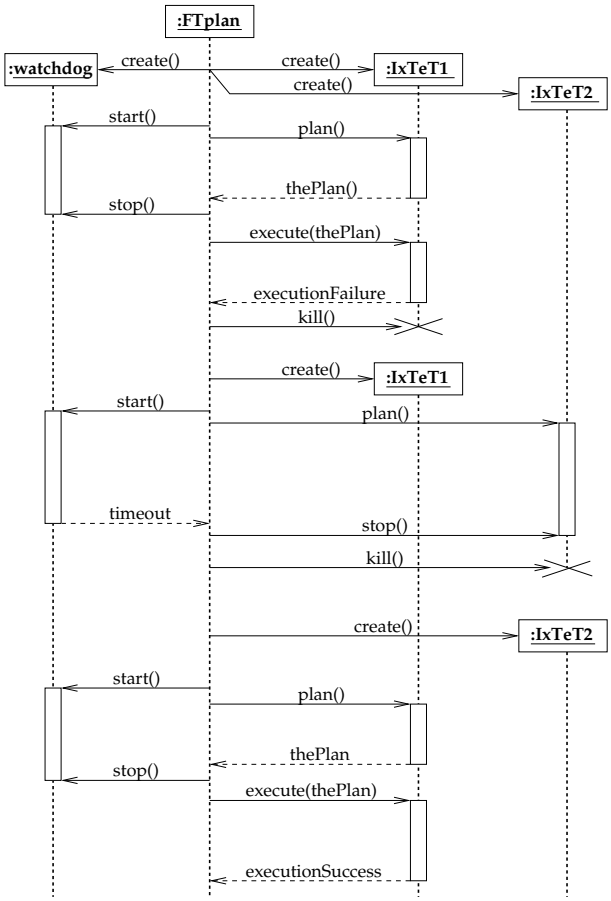

Fig. 6 A fault tolerance scenario with the sequential planning policy

reinitializes the second planner before switching back to the first planner and asking for a new plan. The plan is then produced and successfully executed.

3.2.3 Model diversification

In our implementation, two different planning models are used with the IxTeT planning engine. The first model (which we will call $Model_1$) was originally developed to validate implementation of the LAAS architecture (and particularly the IxTeT component) on a real robot. It is the result of iterative efforts from a different team of researchers. It contains actions needed for a space exploration rover, such as moving to a designated position, photographing objects, and communicating with an orbiter.

A second variant (called $Model_2$) for the same target application has been developed by a different team. Diversification with respect to $Model_1$ has been also forced through specific design choices. For example, robot position is de-

(V : variable ; C : constraint ; E : event ; H : hold)

	$Model_1$				$Model_2$			
	V	C	E	H	V	C	E	H
init. move	2	1	2	2	2	1	1	1
init. camera	2	1	2	2	3	2	2	1
move camera	4	4	2	2	6	5	4	3
take photo	5	4	2	4	4	3	1	4
communicate	3	2	2	3	4	3	1	3
move	19	32	6	5	8	19-45	4	5
	-	-	-	-	8	67-116	5	5

Table 1 Syntactical Comparison between $Model_1$ and $Model_2$

fined using Cartesian coordinates in $Model_1$ whereas $Model_2$ uses a symbolic representation, thus implementing fundamentally different algorithms to those of the Cartesian one. Overall, numerous modifications were carried out, such as pruning redundant system attributes and constraints, or "merging" complementary attributes (e.g., a system position attribute can be combined with a moving/still boolean attribute to give a single attribute that gives the system position when it is motionless, or else the value `MOVING`).

Table 1 presents the syntactical content of each action of the two planning models, showing substantial content differences between the two models. It describes in particular the numbers of variables (V), numerical constraints (C), and event (E) and hold (H) assertions for each action presented in section 3.2.1.

4 Framework for Validation

Our validation framework relies on fault injection at the decisional layer of a full stack of robot controller software, and simulation of robot hardware. Only the robot hardware is simulated, all the software components otherwise execute and interact in real time, in the same way as on a real robot. Although the considered robot controller software stack has been extensively used in demonstrations of a real robot, we preferred to resort to simulation because the behavior of a real robot may become hazardous when we inject faults and it could cause damage to itself or its direct surroundings. A second reason is that numerous repetitive experiments on real robots would be both expensive and hard to automate.

Fault injection is used since it is the *only* way to test the proposed fault tolerance mechanisms with respect to their specific inputs, i.e., faults in planning knowledge. In the absence of any data regarding real faults in declarative models, there is no other practical choice than to rely on *mutations*[2], which have been found to efficiently simulate real faults in imperative languages [13].

We present in this section the framework that has been used to validate the proposed fault tolerant mechanisms: its software architecture, the workload and faultload generated as experiment inputs, the data recorded for each basic

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

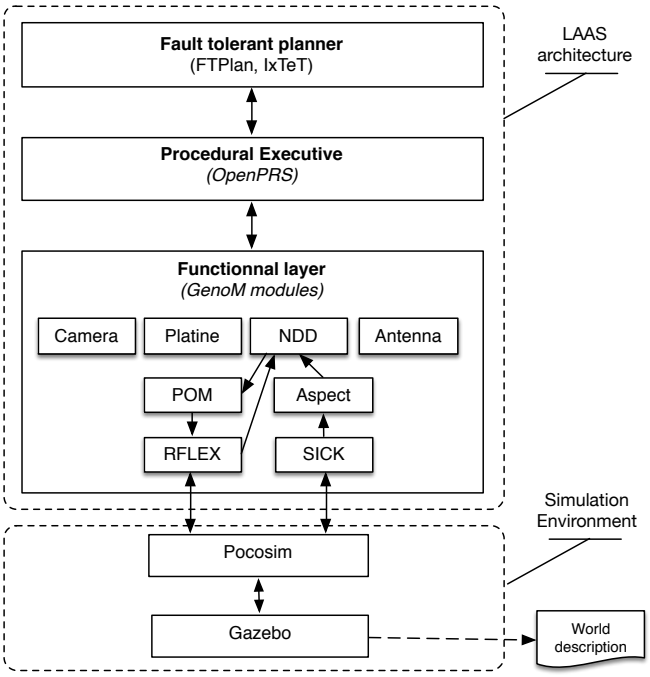

Fig. 7 Simulation environment

experiment and, based on that data, the measurements that represent the output of each set of experiments.

4.1 Testing software architecture

The whole simulation environment is represented in Fig. 7. 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*[3] is used to simulate the physical world and the actions of a mobile robot in that world. It generates realistic sensor feedback and physically plausible interactions between objects through a simulation of rigid-body physics in three dimensions.

The *Pocosim library* [18] is a software bridge between the simulated robot executed on Gazebo and the software commands generated by the GenoM modules.

Our target autonomous system is an existing ATRV (All Terrain Robotic Vehicle) robot commercialized by iRobot, and employs GenoM software modules interfaced with the Gazebo simulated hardware (see Figure 8). The upper

[3] *The player/stage project*, http://playerstage.sourceforge.net

Fig. 8 LAAS-CNRS Dala robot with an ATRV base

layer of the LAAS architecture executes as presented in the previous section. The functional layer consists of eight GenoM modules that can be categorized into three groups:

- The SICK and RFLEX modules both control hardware components through the Pocosim library: SICK controls a laser sensor whereas RFLEX controls wheel motions and an odometer.
- NDD, ASPECT and POM are software modules that use SICK and RFLEX to implement navigation and obstacle avoidance. POM establishes position data of the robot according to the odometer and other possible localization mechanisms. ASPECT uses this position and feedback from SICK to create a map of the robot's immediate surroundings, which is used by NDD to generate navigation commands using a nearness diagram algorithm.
- PLATINE, ANTENNA and CAMERA control hardware components that are not simulated by Gazebo but by software simulation in the GenoM modules themselves (respectively, a camera orientation device, a communication antenna, and two cameras).

4.2 Workload

Autonomous systems move in unpredictable, open and unknown environments. They do not know a priori the obstacles (static or dynamic) that they will encounter, the terrain configuration, the available roads, the presence of external perturbations (weather, lack of brightness, etc.).

Our workload mimics the possible activity of a space rover. The system is required to achieve three subsets of goals during a mission: *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 need to confront the system with several different missions, in several different "worlds". Each *mission* encompasses the number and location of photos to be taken, and the number and occurrence times of

orbiter visibility windows. Each *world* is a set of static obstacles unknown to the robot. These unknown obstacles stress the robot's navigation and obstacle avoidance mechanism (see Section 4.1). At the plan execution level, the unknown obstacles create uncertainty as regards the outcome of action executions, and can possibly prevent the robot from achieving some of its goals.

We implemented four missions and four worlds, thus applying sixteen execution contexts to each fault situation. The missions and worlds are defined with respect to 12m x 12m environment. The initial position of the robot is set equal to the center of this square, at coordinates (0,0). Missions are referenced as gradually more difficult M1 to M4 (Fig. 9): M1 consists in three photos in nearby locations and two communication goals (shown as shaded intervals on the mission time axis), whereas M4 consists in four communications goals and five far apart photo locations. The maximum duration of a mission is 800 seconds, during which it is physically possible to achieve all the objectives. Worlds are referenced as W1 to W4 (Fig. 10). W1 is an empty world with no obstacles to hinder plan execution, while W2 and W3 contain small cylindrical obstacles that are avoidable by our robot navigation and obstacle avoidance mechanism. However, W4 includes larger rectangular obstacles that may be impossible for the navigation module to circumnavigate, and thus susceptible to irremediably block the robot path.

The experiments are inherently non-deterministic, due to asynchrony of the various robot subsystems and in the underlying operating systems. Task scheduling differences between similar experiments may degrade into task failures and possibly unsatisfied goals, even in the absence of faults. To address this non-determinacy, we execute each basic experiment three times, leading to a total of 48 experiments per fault scenario (3 executions $*$ 4 missions $*$ 4 worlds). Ideally More repetition would be needed for statistical inference on the basic experiments, but this would have led to a total number of experiments higher than that which could have been carried out with our available resources (including initialization and data processing, each basic experiment lasts about 20 minutes).

4.3 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 $Model_1$ of Fig. 5).

From a syntactical analysis of the IxTeT formalism, five types of possible mutations were identified:

i) 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.

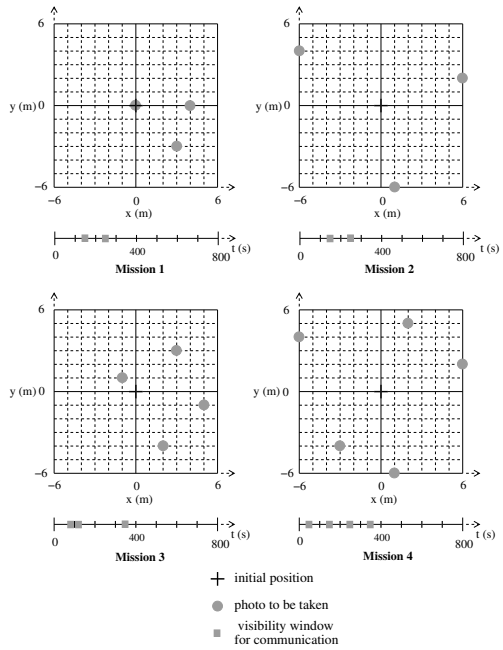

Fig. 9 The Set of Missions: M1 to M4

ii) Substitution of variables: since the scope of a variable is limited to the task where it is defined, numerical (resp. temporal) variables are exchanged successively with all numerical (resp. temporal) variables of the same task.
iii) 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.
iv) 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).
v) 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 [12]. This tool generates a database of possible mutants based on the original source code and a manually generated file called the *mutation table*, which describes all the feasible substitutions. Fig. 11 presents a simple example of a mutation table: each instance of the strings `-oo`, `60` and `1` in the code source will be successively replaced by the strings `-1`, `4` and `26.3`, each substitution generating a particular mutant; each instance of the strings in the set {`?obj`, `?x`, `?y`} will be successively replaced by the other strings in the same set; etc.

Each possible mutant is first compiled off-line, and only added to the database when its compilation does not result in an error, and when its bi-

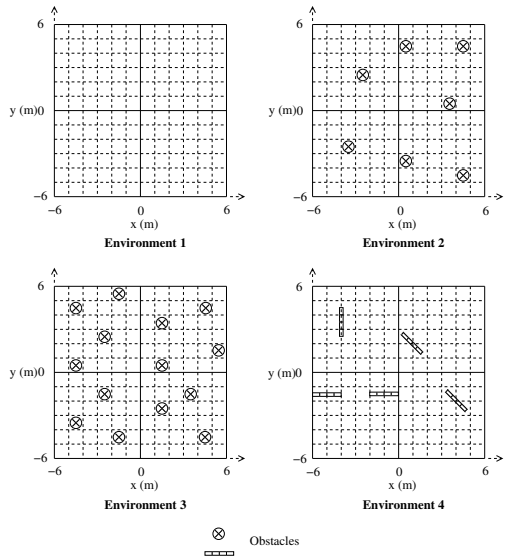

Fig. 10 The Set of Worlds: W1 to W4

```
% substitution of numerical values
{“-oo”, “60”, “1”} → {“-1”, “-4”, “26.3”}

% substitution of variables
{“?obj”, “?x”, “?y”}
{“?t_start”, “?t_end”}

% substitution of attribute values
{“downward”, “straight”, “other”}
{“none”, “done”, “doing”}

% substitution of language operators
{“nonPreemptive”, “latePreemptive”} → {“ “}
{“contingent”} → {“ “}
```

Fig. 11 Mutation Table Example

nary is not identical to that of the non-mutated source code. All in all, more than 1000 mutants were thus automatically generated from the 300 lines planning model. To improve representativeness of injected faults, we also chose to discard mutants where no plan is found in any mission (we consider that models that systematically fail would easily be detected during the development phase).

Finally, to augment the number of relevant experiments, mutant selection was carried out in two phases: (1) a random selection either from the whole database or from a specific type of mutation, (2) a simple manual analy-

sis aimed at eliminating mutants that are trivially equivalent to the original model.

4.4 Recorded data 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, the OpenPRS procedural executive and FTplan; requests and reports sent and received by each planner, as well as outputs of the planning process. For each basic experiment, these uncompressed text files require from 4 to 16 Mb; the 48 experiments characterizing one mutant require nearly 320 Mb.

Condensing this amount of data into significant relevant measures is problematic. Moreover, contrary to more classic mutation experiments, the result of an experiment cannot be easily dichotomized 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. Additionally, detection of equivalent mutants is becoming more complex due to the non-deterministic execution context of autonomous systems.

To address these issues, we chose to categorize the quality of the result of an experiment according to: (a) *mission dependability*, defined in terms of the goals that have been successfully achieved (or alternatively, through the inverse notion of mission *un*-dependability), (b) *mission performance* indicators such as the mission execution time and the distance covered by the robot to achieve its goals, and (c) internal measures of *planning behavior*.

Considering the sets of missions $\mathcal{M}$ given to the system, worlds $\mathcal{W}$ in which it evolved and faults $\mathcal{F}$ injected into the system, mission *un*-dependability for an elementary experiment is given by:

- $\overline{\varphi}_p(\mathcal{M}, \mathcal{W}, \mathcal{F})$, the average proportion of unachieved photo goals,
- $\overline{\varphi}_c(\mathcal{M}, \mathcal{W}, \mathcal{F})$, the average proportion of unachieved communication goals,
- $\overline{\varphi}_r(\mathcal{M}, \mathcal{W}, \mathcal{F})$, the average proportion of unachieved returns to the initial position,
- $\overline{\mu}(\mathcal{M}, \mathcal{W}, \mathcal{F})$, the average proportion of failed missions, where a mission is pessimistically defined as failed if any goal (photo, communication, return) is not achieved.

For example, $\overline{\varphi}_p(M3, W4, 39)$ represents the mean proportion of failed photos for the mission $M3$, in the world $W4$, with the injected fault 39, averaged over the several elementary experiments (currently three) carried out with each injected fault. We define $\mathcal{M}^* = \{M1, M2, M3, M4\}$ (and, similarly, $\mathcal{W}^* = \{W1, W2, W3, W4\}$), such that $\overline{\varphi}_p(\mathcal{M}^*, W4, \emptyset)$ represents the mean proportion of failed photos for all four missions in world $W4$, with no injected faults; it characterizes twelve elementary experiments.

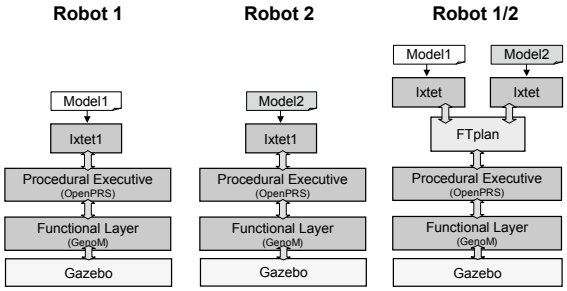

Fig. 12 Experimental Systems: $Robot_1$, $Robot_2$ and $Robot_{1/2}$

We define two measures to characterize the performance of the rover during its mission:

- $\overline{D}(\mathcal{M}, \mathcal{W}, \mathcal{F})$, the average distance (in meters) covered by the rover,
- $\overline{T}(\mathcal{M}, \mathcal{W}, \mathcal{F})$, the average duration (in seconds) of rover activity, i.e., the time at which it performed its last action.

Finally, to characterize the internal behavior of plan execution, we define:

- $\overline{R}(\mathcal{M}, \mathcal{W}, \mathcal{F})$, the average number of replannings during an experiment.

5 Results

Experiments were executed on i386 Linux-based systems with a 3.2 GHz CPU and the Linux OS. We first study the performance cost of the proposed mechanisms, then give three examples of fault injection results, and finally present global results of the fault injection campaign.

We particularly focus on three systems (Fig. 12): a non-redundant robot using $Model_1$ (referred to as $Robot_1$), another non-redundant robot using $Model_2$ (referred to as $Robot_2$) and a diversely redundant robot using FTplan with $Model_1$ and $Model_2$ (referred to as $Robot_{1/2}$).

Two further systems are considered in Section 5.1 when analyzing the performance impact of our mechanisms. Both implement a version of IxTeT with no plan repair capability, thus leading to increased numbers of replannings and model switches: $Robot_1^*$ uses $Model_1$, and $Robot_{1/1}^*$ uses FTplan with the same $Model_1$ for both planners.

5.1 Fault-free performance

To determine the overhead of the proposed fault tolerance mechanisms, we first concentrate on the supposed fault-free models $Model_1$ and $Model_2$. Fig. 13 shows bar-graphs of the observed mission *un*-dependability measures $\overline{\varphi}_p$, $\overline{\varphi}_c$, $\overline{\varphi}_r$, $\overline{\mu}$ and the planning behavior measure $\overline{R}$, for $Robot_1$, $Robot_2$ and $Robot_{1/2}$

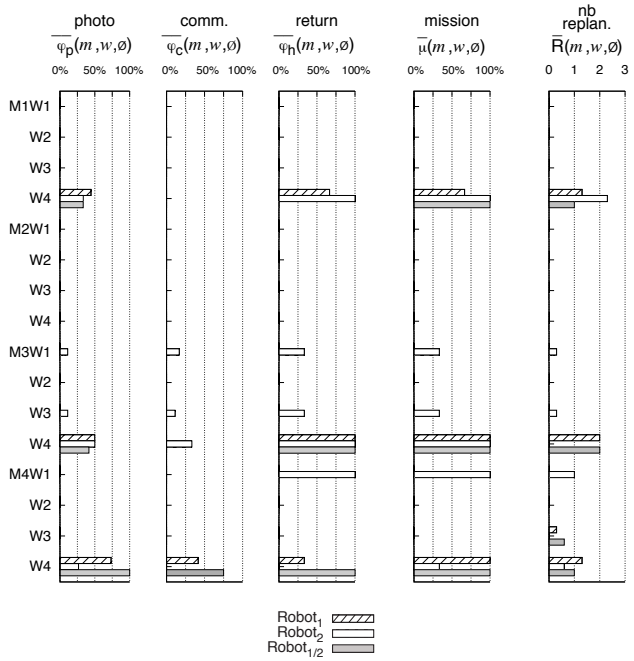

Fig. 13 Mission Undependability and Replannings without Injected Faults

in each of the sixteen mission-world pairs M1W1 to M4W4. Fig. 14 gives bargraphs of the observed performance measures $\overline{D}$ and $\overline{T}$.

Note that results in W4 must be treated with caution: as previously explained, this world contains large obstacles that may cause navigational failures and forever block the robot path, irrespectively of replanning or model switching. Since our work focuses 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 planning 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.

W4 set aside, results are globally very good: $Robot_1$ and $Robot_{1/2}$ succeed in all their goals, while $Robot_2$ 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 $\overline{T}(\mathcal{M}^*, \mathcal{W}^*, \emptyset)$ is 404 seconds for $Robot_1$, 376 seconds for $Robot_2$, and 405 seconds for $Robot_{1/2}$. Time performance-wise, the three systems are thus roughly equivalent.

Although the results are mostly positive, showing that FTplan's main execution loop does not severely decrease goal achievement or performance in the

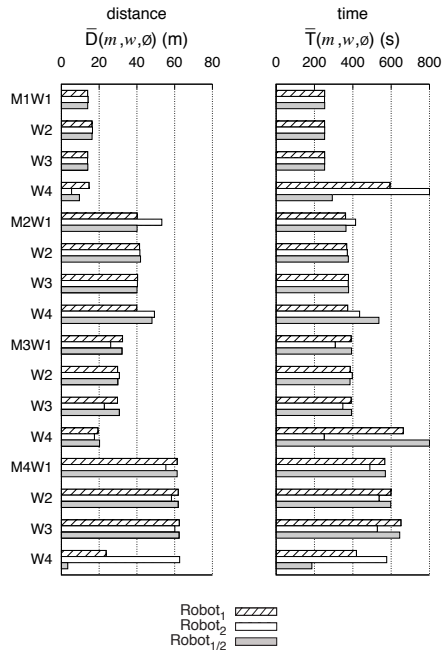

Fig. 14 Mission Performance without Injected Faults

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 using $Robot_1^*$ and $Robot_{1/1}^*$. These two systems use a planner without the optimizing functionality of plan repair (presented in Section 2.2) and are thus particularly prone to time-costly replannings: any failed action systematically leads to complete replanning, with an additional model switch for $Robot_{1/1}^*$. Model 1 was used as both alternatives in $Robot_{1/1}^*$ in order to focus on the cost of the coordination mechanisms rather than on actual recovery. Results are presented in Fig. 15 and Fig. 16.

We effectively see that there are many more replannings (and thus model switches) than in the previous experiments (note the change in scale for $\overline{R}$). The mean number of replannings per experiment is 8.3 for $Robot_1^*$ compared to 0.3 for $Robot_1$, and 8.9 for $Robot_{1/1}^*$ compared to 0.4 for $Robot_{1/2}$. M1W2 appears as a singularity for $Robot_{1/1}^*$: after a few minutes of execution, the IxTeT planner finds no solution in its current situation. We believe that this is due to an elusive bug in either the model, FTplan, or the IxTeT planner. However, the same experiment with $Robot_{1/2}^*$ (using diversification through the first and second models, cf. Figs. 13- 14) gives successful missions, suggesting that the bug lies in the no-plan-repair version of IxTeT that we developed specifically for this experiment.

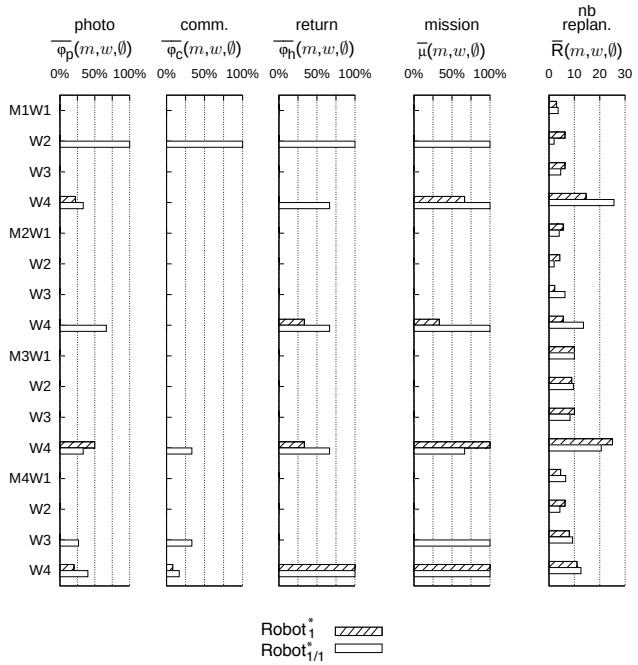

Fig. 15 Mission Undependability and Replannings without Injected Faults and with No Plan Repair

Apart from this singularity, $Robot^*_{1/1}$ only fails more goals than $Robot^*_1$ in the over-stressing W4 execution contexts, as well as the quite complex mission-world pair M4W3. Setting aside W4 (and the M1W2 singularity), the mean activity time $\overline{T}$ is 381 seconds for $Robot^*_1$ and 431 seconds for $Robot^*_{1/1}$, indicating an overhead of 13%. Including W4, $\overline{T}$ is 456 seconds for $Robot^*_1$ and 535 seconds for $Robot^*_{1/1}$, indicating an overhead of 17%. We deem these results to be quite acceptable considering the negative impact of discarding plan repair.

5.2 Fault-tolerance efficacy

We present here results regarding $Robot_1$ and $Robot_{1/2}$, when the same faults are injected in $Model_1$ of both systems. We first begin with three mutation examples, before detailing global results for the 28 mutants considered in our experiment campaign.

5.2.1 Results for three specific mutations

Three examples of mutations are described here, to present the type of results and behavior that occurred in our experiments. For a more compact display, only mission undependability and planning behavior measures are represented.

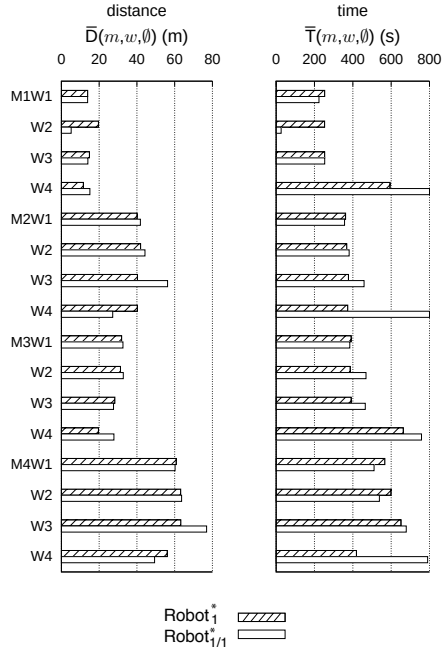

Fig. 16 Mission Performance without Injected Faults and with No Plan Repair

Results for the first mutation, with identifier 39, are presented in Fig. 17. This mutation causes an overly constrained robot position during a camera shot. It thus results in the planner being unable to find plans for missions where photographs have to be taken in positions that do not respect the overly-restrictive constraint (this is the case for missions M2 and M4). This example illustrates the significance of the mission-world pairs chosen as validation scenarios: many different missions need to be considered since faults can remain inactivated in some missions. Since testing the practically infinite execution context is well nigh impossible, this example also underlines the difficulty of testing and thus the relevance of fault tolerance mechanisms to address residual faults in the deployed planning models.

Fig. 18 presents the results for mutation 589. The fault injected in this mutation affects only the movement recovery action of the planning model. Thus, contrary to the previous example, correct plans are established for all missions. However, as soon as a movement action fails, the planner is unable to find a plan allowing recovery of the movement, which causes failure of the system. This is particularly obvious in the case of missions M1 and M3, where short distances between photograph locations lead to a short temporal margin for the movement action. As acceleration, deceleration and rotation are not considered in the planning model, movements are susceptible to take longer than estimated, and can thus be interrupted by the plan execution controller and considered failed, necessitating a recovery action. In missions M2 and

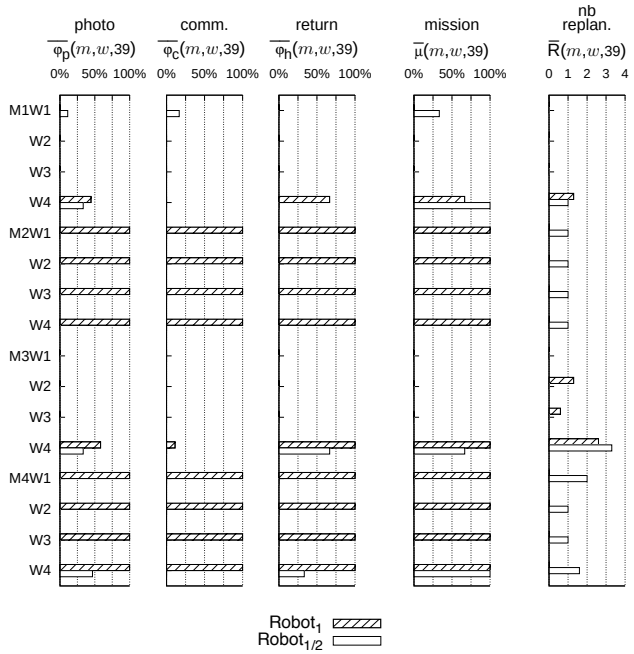

Fig. 17 Results for mutation 39

M4, movements cover greater distances, resulting in larger temporal margins and thus fewer movement action failures. $Robot_{1/2}$ tolerates this fault to some extent: completely in mission M2 and partially in mission M4. The high failure rate of mission M3 for $Robot_{1/2}$ can be explained by a domino effect due to communication goals being given priority over photography goals. When the fault is activated due to a failed movement action, FTplan switches to $Model_2$ and requests a plan. However, a communication goal is now so near that the planner is unable to find a plan to achieve it, so it abandons goals of lesser priority, but to no avail. This example leads to two important observations:

- First, testing with many different missions and worlds is once again underlined, as the fault is not activated in several scenarios.
- Second, testing must be carried out in an *integrated system*. Indeed, the original plans produced by the planner are correct, as well as the lower levels of the system. However, the planning model contains a serious fault that can cause critical failure of the system in executions where recovery is required.

The results of mutation 583 are presented in Fig. 19. It is the only case in our 28 experiments where the fault intolerant $Robot_1$ shows *better* results than the fault tolerant $Robot_{1/2}$, although we identified 8 other mutations where results in both systems were similar (including five mutants suspected to be equivalent to the original model). The injected fault causes the duration of the robot movement to be underestimated in plans, resulting in execution errors

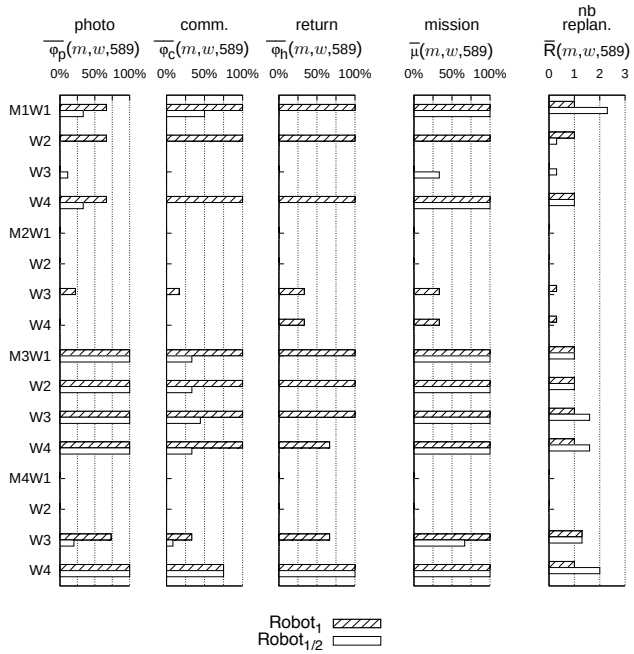

Fig. 18 Results for mutation 589

and replannings that lead to some goals being missed. Failures of $Robot_{1/2}$ are not directly linked to the injected fault, but rather to the use of the poorly optimized second model. It uses more pessimistic time constraints than the first model, thus giving up some goals, and causing failure of all photographs in mission M4 through a similar domino effect to that presented in the previous example.

5.2.2 Overall results for all mutations

To assess the overall efficacy of the proposed mechanisms and the FTplan component, we injected 38 faults in our first model, leading to more than 3500 experiments and 1200 hours of testing. To be conservative, we discarded 10 mutants that were unable to find a plan for any of the four missions[4]. 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, leading to a pessimistic estimation of the improvement offered by FTplan.

Our experimental faultload (noted $\mathcal{F}^*$) thus consists of 28 mutations, corresponding to: three substitutions of attribute values, six substitutions of

[4] In these cases, *Robot*1 obviously fails every goal, while $Robot_{1/2}$ gives the same results as a fault-free $Robot_2$: a perfect success rate for most of the missions.

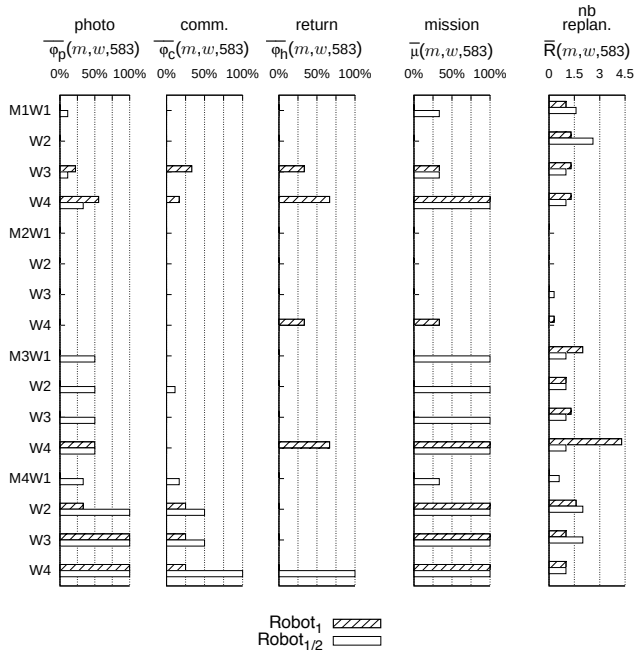

Fig. 19 Results for mutation 583

Table 2 Decrease of Goal Failure Proportions Using FTplan (missions $\mathcal{M}^*$, faults $\mathcal{F}^*$)

	photos $\delta_{\overline{\varphi}_p}$	comms $\delta_{\overline{\varphi}_c}$	returns $\delta_{\overline{\varphi}_r}$	missions $\delta_{\overline{\mu}}$
$\mathcal{W}^*\backslash$W4	62%	70%	80%	41%
$\mathcal{W}^*$	50%	64%	58%	29%

variables, ten substitutions of numerical values, four substitutions of operators, and six removals of constraints. The mutants were executed on $Robot_1$ and $Robot_{1/2}$. The overall results of our fault injection campaign are presented in Fig. 20. The results are averaged over all four missions (set $\mathcal{M}^* = \{$M1,M2,M3,M4$\}$) and all four worlds (set $\mathcal{W}^* = \{$W1,W2,W3,W4$\}$). Results are also given when excluding the unforgiving world W4 (set $\mathcal{W}^*\backslash$W4). The figure also recapitulates the results in the absence of faults (i.e., $\mathcal{F} = \emptyset$) (already presented in Fig. 13) which notably show that, even *without* injected faults, $Robot_1$ and $Robot_{1/2}$ are unable to meet some goals in world W4. Table 2 presents the improvement procured by FTplan measured in terms of the percentage decrease of goal failure proportions, that is: $\delta_{\overline{\mathcal{Z}}} = (\overline{\mathcal{Z}}(Robot_1) - \overline{\mathcal{Z}}(Robot_{1/2}))/\overline{\mathcal{Z}}(Robot_1)$ with $\overline{\mathcal{Z}} \in \{\overline{\varphi}_p, \overline{\varphi}_c, \overline{\varphi}_r, \overline{\mu}\}$.

These results show that, with the considered faultload:

– The redundant diversified models of the fault-tolerant $Robot_{1/2}$ provide a notable improvement to dependability in the presence of faults: in all cases,

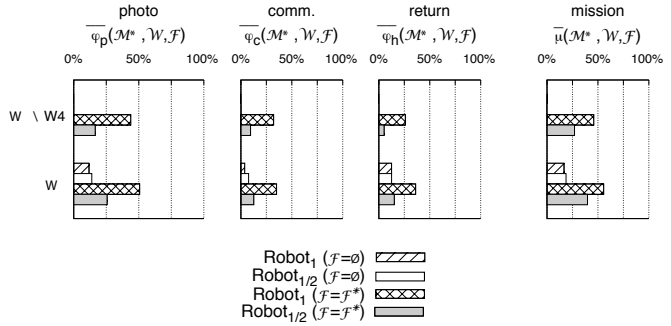

Fig. 20 Impact of planner redundancy in presence of faults

the proportions of failed goals decrease compared to the non-redundant $Robot_1$.

– Even when considering the pessimistic measure of the proportion of failed missions (recall that a mission is considered as failed even if only a single elementary goal is not achieved), the improvement procured by redundant diversified models is appreciable: 41% in worlds W1-W3, 29% when world W4 is also considered.

Note, however, that in the presence of injected faults, the fault-tolerant $Robot_{1/2}$ is *less* successful than a single fault-free model (compare $Robot_{1/2}(\mathcal{F} = \mathcal{F}^*)$ with $Robot_1(\mathcal{F} = \emptyset)$ on Fig. 20). 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.

6 Conclusion and perspectives

Lack of dependability remains a severe impediment to the take-up and practical utilization of autonomous systems. At the software level, the dependability of decisional mechanisms such as planners, which are essential for truly autonomous operation, is particularly challenging. The difficulty of validating such autonomy software makes it very difficult to provide safety and reliability guarantees for autonomous systems.

In this paper, we focused on the reliability aspect of the problem, proposing an innovative fault tolerance approach for temporal planners. The proposed approach aims to complement verification and testing by providing tolerance to residual design faults in coded domain-specific knowledge. To this end, we advocate the use of diversified planning models and search heuristics. 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. It would also be worthwhile to assess the impact of diversification in planning heuristics rather than just models, and study the possible benefits of using more than two diversified versions. An interesting point would also be to keep traces of success scores of the planners to define a preferred order for planners (e.g., invoke less successful planners when the "best" planners have failed). In addition, the statistical relevance of the results would benefit from many more experiments. 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. Alami, R., Chatila, R., Fleury, S., Ghallab, M., Ingrand, F.: An Architecture for Autonomy. International Journal of Robotics Research **17**(4), 315–337 (1998)
2. Armoush, A.: Design patterns for safety-critical embedded systems. Ph.D. thesis, Embedded Software Laboratory - RWTH Aachen University (2010)
3. Avižienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic Concepts and Taxonomy of Dependable and Secure Computing. IEEE Trans. on Dependable and Secure Computing **1**(1), 11–33 (2004)
4. Bernard, D.E., Gamble, E.B., Rouquette, N.F., Smith, B., Tung, Y.W., Muscettola, N., Dorias, G.A., Kanefsky, B., Kurien, J., Millar, W., Nayal, P., Rajan, K., Taylor, W.: Remote Agent Experiment DS1 Technology Validation Report. Ames Research Center and JPL (2000)
5. Bouguerra, A., Karlsson, L., Saffiotti, A.: Monitoring the execution of robot plans using semantic knowledge. Robotics and Autonomous Systems **56**(11), 942 – 954 (2008)
6. Brat, G., Denney, E., Giannakopoulou, D., Frank, J., Jonsson, A.: Verification of Autonomous Systems for Space Applications. In: IEEE Aerospace Conference 2006. Big Sky, Montana (2006)
7. Cesta, A., Finzi, A., Fratini, S., Orlandini, A., Tronci, E.: Validation and verification issues in a timeline-based planning system. The Knowledge Engineering Review **25**(Special Issue 03), 299–318 (2010)
8. Chen, I.R.: Effects of Parallel Planning on System Reliability of Real-Time Expert Systems. IEEE Transactions on Reliability **46**(1), 81–87 (1997)
9. Chen, I.R., Bastani, F.B., Tsao, T.W.: On the Reliability of AI Planning Software in Real-Time Applications. IEEE Transactions on Knowledge and Data Engineering **7**(1), 14–25 (1995)

10. Chien, S., Sherwood, R., Tran, D., Cichy, B., Rabideau, G., Castano, R., Davis, A., Mandl, D., Trout, B., Shulman, S., Boyer, D.: Using autonomy flight software to improve science return on earth observing one. Journal of Aerospace Computing, Information, and Communication **2**(4), 196–216 (2005)
11. Crestani, D., Godary-Dejean, K., Lapierre, L.: Enhancing fault tolerance of autonomous mobile robots. Robotics and Autonomous Systems **68**(0), 140 – 155 (2015)
12. Crouzet, Y., Waeselynck, H., Lussier, B., Powell, D.: The SESAME Experience: from Assembly Languages to Declarative Models. In: Proceedings of the 2nd Workshop on Mutation Analysis. Raleigh, NC (2006)
13. Daran, M., Thévenod-Fosse, P.: 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 (1996)
14. Ghallab, M., Laruelle, H.: Representation and control in IxTeT, a temporal planner. In: The Second Artificial Intelligence Planning Systems Conference, pp. 61–67. AAAI Press, Chicago, IL, USA (1994)
15. Goldberg, A., Havelund, K., McGann, C.: Runtime Verification for Autonomous Space Craft Software . In: IEEE Aerospace Conference 2005. Big Sky, Montana (2005)
16. Havelund, K., Lowry, M., Penix, J.: Formal Analysis of a Space Craft Controller using SPIN. IEEE Transactions on Software Engineering **27**(8), 749–765 (2001)
17. Howey, R., Long, D., Fox, M.: VAL: Automatic Plan Validation, Continuous Effects and Mixed Initiative Planning using PDDL. In: ICTAI. Boca Raton, Florida (2004)
18. Joyeux, S., Lampe, A., Alami, R., Lacroix, S.: Simulation in the LAAS Architecture. In: Software Development in Robotics Workshop, International Conference on Robotics and Automation (ICRA05). Barcelona, Spain (2005)
19. Khatib, L., Muscettola, N., Havelund, K.: Mapping Temporal Planning Constraints into Timed Automata. In: TIME, pp. 21–27. Cividale del Friuli, Italy (2001)
20. Lambèr, R., Rinie, v.E.: A literature review on new robotics: Automation from love to war. International Journal of Social Robotics pp. 1–22 (2015)
21. Lemai, S., Ingrand, F.: Interleaving Temporal Planning and Execution in Robotics Domains. In: The National Conference On Artificial Intelligence, pp. 617–622. San Jose, California (2004)
22. Lussier, B., Gallien, M., Guiochet, J., Ingrand, F., Killijian, M.O., Powell, D.: Fault Tolerant Planning for Critical Robots. In: Dependable Systems and Networks (DSN07). Edinburgh, UK (2007)
23. Lussier, B., Gallien, M., Guiochet, J., Ingrand, F., Killijian, M.O., Powell, D.: Planning with diversified models for fault-tolerant robots. In: Proc. of The International Conference on Automated Planning and Scheduling (ICAPS07), Providence, Rhode Island, USA, pp. 216–223 (2007)
24. Lussier, B., Lampe, A., Chatila, R., Ingrand, F., Killijian, M.O., Powell, D.: 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 (2005)
25. Machin, M., Dufossé, F., Blanquart, J., Guiochet, J., Powell, D., Waeselynck, H.: Specifying safety monitors for autonomous systems using model-checking. In: B. Andrea, D.G. Felicita (eds.) The 33rd International Conference on Computer Safety, Reliability and Security (SAFECOMP2014), pp. 262–277. Springer International Publishing (2014)
26. Mendoza, J.P., Veloso, M., Simmons, R.: Mobile robot fault detection based on redundant information statistics. In: Workshop at IROS'12 on "Safety in human-robot coexistence and interaction: How can standardization and research benefit from each other?", Vilamoura, Portugal (2012)
27. Menzies, T., Pecheur, C.: Verification and validation and artificial intelligence. Advances in Computers **65**, 153 – 201 (2005)
28. Monterlo, M., Thrun, S., Dahlkamp, H., Stavens, D., Strohband, S.: Winning the DARPA Grand Challenge with an AI Robot. In: American Association of Artificial Intelligence 2006 (AAAI06). Boston, MA (2006)
29. Muscettola, N., Dorais, G.A., Fry, C., Levinson, R., Plaunt, C.: IDEA: Planning at the Core of Autonomous Reactive Agents. In: AIPS 2002 Workshop on On-line Planning and Scheduling. Toulouse, France (2002)

30. Penix, J., Pecheur, C., Havelund, K.: Using Model Checking to Validate AI Planner Domain Models. In: SEW. Greenbelt, Maryland (1998)
31. Pettersson, O.: Execution monitoring in robotics: A survey. Robotics and Autonomous Systems **53**(2), 73 – 88 (2005)
32. Randell, B.: System Structure for Software Fault Tolerance. IEEE Transactions on Software Engineering **1**, 220–232 (1975)
33. Sarkar, D., Dubey, S.K., Mahapatra, A., Roy, S.S.: Modeling and analysis of fault tolerant gait of a multi-legged robot moving on an inclined plane. Procedia Technology **14**(0), 93 – 99 (2014)
34. Urmson, C., Anhalt, J., Bae, H., Bagnell, J.A., Baker, C.R., Bittner, R.E., Brown, T., Clark, M.N., Darms, M., Demitrish, D., Dolan, J.M., Duggins, D., Ferguson , D., Galatali, T., Geyer, C.M., Gittleman, M., Harbaugh, S., Hebert, M., Howard, T., Kolski, S., Likhachev , M., Litkouhi, B., Kelly , A., McNaughton, M., Miller, N., Nickolaou, J., Peterson, K., Pilnick, B., Rajkumar, R., Rybski, P., Sadekar, V., Salesky, B., Seo, Y.W., Singh, S., Snider, J.M., Struble, J.C., Stentz, A., Taylor , M., Whittaker, W.R.L., Wolkowicki, Z., Zhang, W., Ziglar, J.: Autonomous driving in urban environments: Boss and the urban challenge. Journal of Field Robotics Special Issue on the 2007 DARPA Urban Challenge, Part I **25**(8), 425–466 (2008)
35. Zaman, S., Steinbauer, G., Maurer, J., Lepej, P., Uran, S.: An integrated model-based diagnosis and repair architecture for ROS-based robot systems. In: Robotics and Automation (ICRA), 2013 IEEE International Conference on, pp. 482–489 (2013)
36. Zhang, Y., Jiang, J.: Bibliographical review on reconfigurable fault-tolerant control systems. Annual Reviews in Control **32**(2), 229 – 252 (2008)