Planning with Diversified Models for Fault-Tolerant 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

Planners are central to the notion of complex autonomous systems. They provide the flexibility that autonomous systems need to be able to operate unattended in an unknown and dynamically-changing environment. However, they are notoriously hard to validate. This paper reports an investigation of how redundant, diversified models can be used as a complement to testing, in order to tolerate residual development faults. A fault-tolerant temporal planner has been designed and implemented using diversity, and its effectiveness demonstrated experimentally through fault injection. The paper describes the implementation of the fault-tolerant planner and discusses the results obtained. The results indicate that diversification provides a noticeable improvement in planning reliability with a negligible performance overhead. However, further improvements in reliability will require implementation of an on-line checking mechanism for assessing plan validity before execution.

Introduction

Planning shows promising success as a central decisional mechanism in complex autonomous systems, both in space exploration and in experimental studies. However, the dependability of planners remains a stumbling block to real life utilization. Indeed, how can we justifiably trust such mechanisms, whose behavior is difficult to predict and validate? Autonomous systems strive to accomplish goals in open environments. The space of possible execution contexts is thus, in essence, infinite, and cannot be exhaustively tested during validation. Testing and other validation techniques are nevertheless necessary in order to obtain planners that are as bug-free as possible. However, we believe that such validation techniques must be complemented by a fault-tolerance approach aimed at making planners resilient to residual bugs. We propose in this paper a fault tolerance approach focused on development faults in planner knowledge. The approach uses redundant diversified planning models.

First, we introduce basic concepts of dependability and dependability issues relative to planning. Second, we present fault tolerance techniques that may be applied to planning mechanisms and the implementation of a fault tolerant planner for an autonomous system architecture. Third, we introduce the validation framework that we developed to assess performance and efficacy of our fault tolerance approach. Finally, we present our experimental results.

Dependability and Fault Tolerance

Dependability is a major concern in computing systems controlling critical structures such as railroads, planes and nuclear plants. We introduce here basic dependability concepts extracted from (Avizienis *et al.* 2005). We then discuss validation and other dependability issues relative to planning.

Dependability basic concepts

The *dependability* of a computing system is its ability to deliver service that can justifiably be trusted. *Correct service* is delivered when the service implements the system *function*, that is what the system is intended to do. Three concepts further describe this notion: the attributes of dependability, the threats to dependability, and the means by which dependability can be attained (Figure 1).

Dependability encompasses numerous attributes. Depending on the application intended for the system, different emphasis may be put on each attribute. We focus particularly on *reliability* (the continuous deliverance of correct service for a period of time) and *safety* (the absence of catastrophic consequences on the users and the environment) of a system.

The *threats* to a system's dependability consist of failures, errors and faults. A system *failure* is an event that occurs when the delivered service deviates from correct service. An *error* is that part of the system state that can cause a subsequent failure. An error is detected if its presence is indicated by an error message or error signal. A *fault* is the adjudged or hypothesized cause of an error. Here, we focus particularly on *development faults*: faults that are unintentionally caused by man during the development of the system.

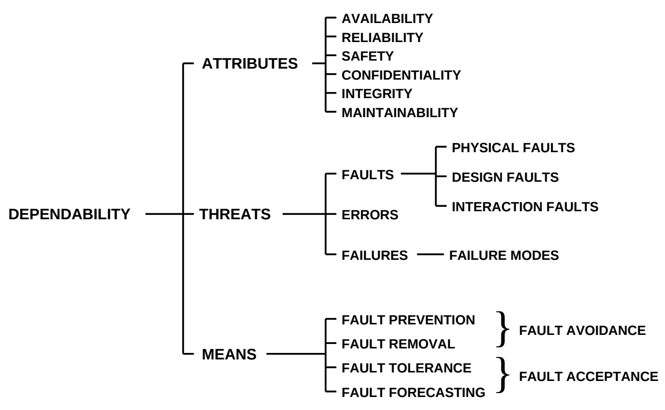

Figure 1: Dependability tree (Avizienis *et al.* 2005)

The *means* to make a system dependable are regrouped in four concepts: (a) *fault prevention*, that is how to prevent the occurrence or introduction of faults, (b) *fault removal*, that is how to reduce the number or severity of faults, (c) *fault tolerance*, that is how to deliver correct service in the presence of faults, and (d) *fault forecasting*, that is how to estimate the present number, the future incidence, and the likely consequences of faults. Fault prevention and fault removal can together be considered as *fault avoidance*, that is the attempt to develop a system without faults. Fault tolerance and fault forecasting embody the concept of *fault acceptance*, in that they attempt to estimate and reduce the consequences of the remaining faults, knowing that fault avoidance is almost inevitably imperfect. The development of a dependable computing system calls for the combined utilization of these four techniques.

Planning Validation and other Dependability Issues

Currently, planning dependability relies almost exclusively on fault avoidance, with only few attempts to consider fault acceptance as a complementary approach.

Planner development tools such as Planware (Becker & Smith 2005) provide a level of fault prevention up front in that they assist domain analysis and knowledge acquisition by use of a high-level graphical formalism. However, it is the notorious problem of planner validation (i.e., fault removal) that has received the most attention to date, especially in autonomous systems.

Indeed, the 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*, 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. 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).

The planning engine itself is not much different from any other software, so traditional software verification techniques may be applied. For example, (Brat *et al.* 2006) report successful use of static analysis of a planner engine using the PolySpace C++ Verifier. There has also been work on model-checking planner executives (systems that execute plans) that successfully identified serious concurrency problems in the Deep-Space One spacecraft (Havelund, Lowry, & Penix 2001).

But the trickiest part of planner software is most certainly the domain model itself, since it is this that changes from one application to another and that *in fine* dictates what plans are produced. Typically, a planning domain model and the associated domain-specific search heuristics are constructed and refined incrementally, by testing them against a graded set of challenges (Goldberg, Havelund, & McGann 2006). However, a small change to either can have surprisingly dramatic changes in the planners 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 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 (Bernard *et al.* 2000), and is supported by the VAL validation tool (Howey, Long, & Fox 2004). 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 that are included in the model, this approach only really provides confidence about the planning engine and not the domain-specific model.

Automatic static analysis may also be used to ascertain properties on planning models, allowing detection of inconsistencies and missing constraints, whereas manual static analysis requires domain experts to closely scrutinize models proposed by planning developers. The development tool Planware (Becker & Smith 2005) offers facilities for both types of analysis.

Some work (Khatib, Muscettola, & Havelund 2001; Penix, Pecheur, & Havelund 1998) has attempted to validate domain-specific models by means of model-checking approaches. However, (Goldberg, Havelund, & McGann 2006) reports scalability issues when the models become large, and suggests instead run-time verification of plan execution traces against high-level

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

properties expressed in temporal logic. However, no details are given about how such high-level properties are defined.

To the best of our knowledge, not much work has been done on fault acceptance in planners. Whereas planner models commonly consider faults affecting domain objects (e.g., hardware faults affecting sensors or actuators), residual development faults affecting the planner itself are seldom considered.

One exception is (Chen, Bastani, & Tsao 1995), which proposes a measure for planner software reliability and 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 (Chen 1997) addresses this compromise through a fault-tolerance approach based on concurrent use of planners with diversified heuristics: a first heuristic, quick but dirty, 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 an essential complement to verification and testing for planners embedded within critical autonomous systems.

Fault Tolerant Planning

Complementary to testing, diverse redundancy is the only known approach to improve trust in the behavior of a critical system regarding residual development faults (e.g., diversification is used in software components of the Airbus A320, and in hardware components of the Boeing B777). We propose here fault tolerant techniques based on diversity, adapted to planners and focusing on development faults in the planner knowledge. We first introduce the general principles of the proposed mechanisms, and then present an implementation in an existing autonomous system architecture.

Principles

The general principle of the mechanisms that we propose is similar to that of recovery blocks (Randell 1975) and distributed recovery blocks (Kim & Welch 1989). Diversified variants of the planner are executed either sequentially or concurrently. A recovery choice is made according to errors detected in the plans produced. Diversity between the variants is encouraged by forcing the use of different algorithms, variable domains and parameters in the models, and different heuristics of the variants.

Detection Implementing error detection for decisional mechanisms in general, and planners in particular, is difficult. Indeed, 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 search engine.

A plan failure detector is a classical mechanism used in robotics for execution control. Failure of an action that 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, the search 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. This implies that the checker maintains an internal representation of the system state and of the goals that have been reached. Goals can only be declared as failed when every action of the plan has been carried out.

Recovery We propose two recovery mechanisms, based on redundant planners using diverse knowledge. Theoretically, redundant planners could be used to tolerate not only faults in planning models but also in the search engine. However, no practical work has been done to test this possibility.

With the first mechanism, the planners are executed *sequentially*, one after another. The principle is given in Figure 2. 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 the previous section: 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

```
1. begin mission
2.   failed_planners $\leftarrow \emptyset$;
3.   while (goals $\neq \emptyset$)
4.     candidates $\leftarrow$ planners;
5.     while (candidates $\neq \emptyset$ & goals $\neq \emptyset$)
6.       choose k such as (k $\in$ candidates)
                      & (k $\notin$ failed_planners);
7.       candidates $\leftarrow$ candidates $\setminus$ k;
8.       init_watchdog(max_duration);
9.       send (plan_request) to k;
10.      wait % for either of these two events
11.        □ receive (plan) from k
12.          stop_watchdog;
13.          if analyze(plan)=OK then
14.             failed_planners $\leftarrow \emptyset$;
15.             k.execute_plan();
                % if the plan fails goals != empty
                % and then we loop to line 5 or 3
16.          else
17.             send(k.invalid_plan) to operator;
18.             failed_planners $\leftarrow$ failed_planners $\cup$ k;
19.          end if
20.         □ watchdog_timeout
21.           failed_planners $\leftarrow$ failed_planners $\cup$ 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 2: Sequential Planning Policy

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* (Lussier 2007). The main differences with respect to the algorithm given in Figure 2 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.

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*.

To avoid error propagation from a possibly faulty planner, FTplan should not rely on any information that comes from or depends on the planners themselves. FTplan maintains thus its own system state representation, based on information obtained from the execution control layer.

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 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 planners it coordinates, we can safely rely on classic verification and testing to assume that it is fault-free.

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.

LAAS Architecture The LAAS architecture is presented in (Alami *et al.* 1998), and some recent modifications have been proposed in (Lemai & Ingrand 2004). It has been successfully applied to several mobile robots, some of which have performed missions in open environments (including exploration and human interaction). It is composed of three main components[2] as presented in Figure 3: 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* is in charge of decomposing and refining plan actions into lower-level actions executable by functional components, and executing them. During execution, OpenPRS reports any action failures to the planner, in order to re-plan or repair the plan.

IxTeT is a temporal constraint planner that builds plans by combining high level actions. 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.

Fault Tolerant Planner Implementation The fault tolerance principles presented previously have been implemented in a fault tolerant planner component as presented in Figure 4. This component replaces the original component “Planner” presented in Figure 3. The FTplan component is in charge of communicating with OpenPRS as the original planner does.

The current version of FTplan implements the sequential redundant planner coordination algorithm presented in Figure 2, 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 earlier: watchdog timer, plan failure detection, and on-line goal checker.

[2]An additional robustness component, R2C, is introduced in (Py & Ingrand 2004). We have not considered it in this study since its current implementation is not compatible with our experimentation environment.

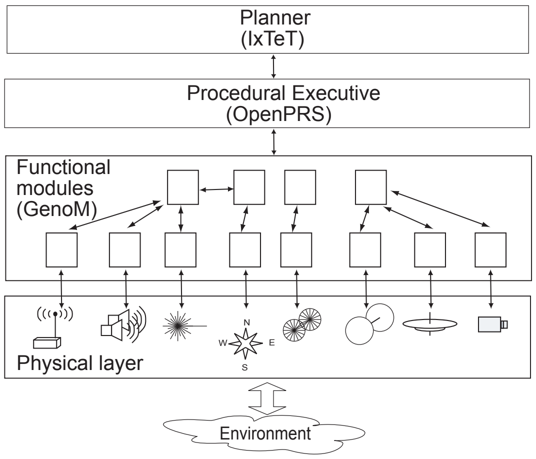

Figure 3: The LAAS architecture

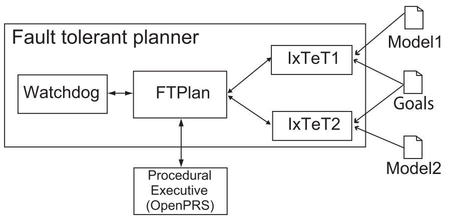

Figure 4: Fault tolerant planner

The watchdog timer is launched at each start of planning. As soon as a plan is found before the assumed worst-case time limit (40 seconds in our implementation), 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 checking is performed after each action executed by OpenPRS that can result in a modification in the goal achievements. 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 current 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.

The first model used by FTplan was thoroughly tested and used on a real ATRV (All Terrain Robotic Vehicle) robot. We thus 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. In the same way, orientation of the robot cameras is characterized symbolically in the first model and numerically in the second. Other forced diversification resulted in merging some of the model attributes (that are variables that together describe the system state in the planning models) and modifying several constraint expressions. Experiments showed that this second model, although less time-performant than the first model, had similar fault-free results considering goal accomplishment for the considered workload.

Experiments and Validation

Our validation framework relies on simulation and fault injection. Simulation (Joyeux *et al.* 2005) is used since it is both safer and more practical to exercise the autonomy software on a simulated robot than on a real one. The *autonomous system* is based on an existing ATRV robot, and employs GenoM software modules interfaced with the simulated hardware. 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 (Daran & Thévenod-Fosse 1996).

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

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.

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

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).

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 4).

Five types of possible mutations were identified from the model syntax (Crouzet *et al.* 2006): (a) substitution of numerical values, (b) substitution of variables, (c) substitution of attribute[4] values, (d) substitution of language operators and (e) removal of constraint relations. 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.

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 and OpenPRS, 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.

Results

We present in this part several experimental results using the evaluation framework previously introduced.

Experiments

Experiments were executed on i386 systems with a 3.2 GHz CPU and Linux OS. Fault-free experiments were first realized to assess the overhead of the proposed fault tolerance mechanisms. They were conclusive in showing that the proposed mechanisms did not severely degrade the system performance in the chosen activities (Lussier *et al.* 2007). These experiments also showed that results in the world 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 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 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.

To test the efficacy of the fault tolerance 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

[4]Remember that, in the IxTeT formalism, attributes are the different variables that together describe the system state.

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

criteria. We thus include the results obtained with these five mutants, leading to a pessimistic estimation of the improvement offered by FTplan.

The 28 considered mutations consist of: three substitutions of attribute values, six substitutions of variables, nine substitutions of numerical values, four substitutions of operators, and six removals of constraints. The mutants were executed on two different robots: Robot1 uses our first model (with one injected fault), and Robot1/2 contains an FTplan component that uses successively our first model (with the same injected fault) and our fault-free second model.

Behavior in the Presence of Faults

We develop here the results of two mutations as examples of our experiments. In each case (Figure 5 and Figure 6), our results present five different measures: (a-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 an experiment (in the case of Robot1/2, this number is equivalent to the number of model switches during the mission).

Results for the first mutation, identified by the reference 1-39, are presented in Figure 5. This mutation causes an overly constrained robot position during a camera shot. It thus results in the planner's inability 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 system activity chosen for the evaluation: numerous different missions are necessary because faults can remain dormant in some missions. Since testing the practically infinite execution context is well nigh impossible, this example underlines the difficulty of testing and thus the interest of fault tolerance mechanisms to cover residual faults in the deployed planning models.

Figure 6 presents the results for mutation 1-589. The fault injected in this mutation affects only a 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 action movement. As acceleration, deceleration and rotation are not considered in the planning model, movements are susceptible to take longer than estimated, and can thus be be interrupted by the execution controler and considered failed, necessitating a recovery action. In missions M2 and M4, movements cover greater distances, resulting in larger temporal margins and thus fewer movement action failures. Robot1/2 tolerates this

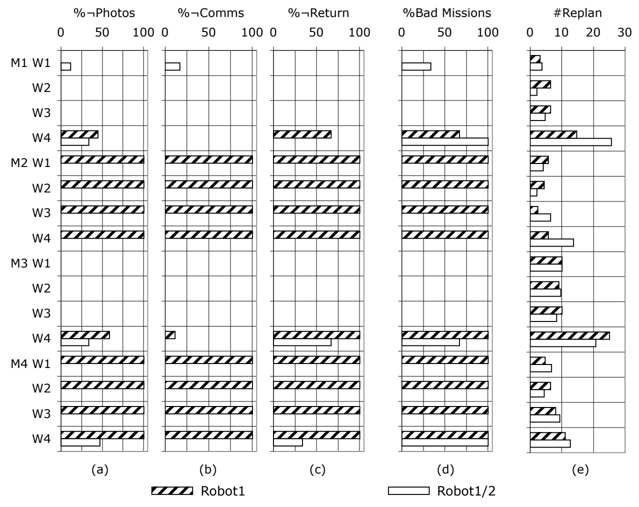

Figure 5: Results for mutation 1-39

fault to some extent: completely in mission M2 and partially in mission M4. The high failure rate of mission M3 for Robot1/2 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 Model2 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 underlines two important issues:

- First, testing with numerous diversified missions and environments is once again pointed out, as the fault is not activated in several activities.
- Second, testing must be realized 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 some executions.

Fault Tolerance Efficacy

The general results, including all 28 mutations, 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 the experiments show that RobotFT in the presence of injected faults is *less* successful than a single fault-free model (some goals are still missed, even when the constrained worl W4 is omitted). This apparent decrease in dependability is explained by the fact that, in our current implementation, incorrect plans are only detected when their execution has at least partially failed, possibly rendering one or more goals unachievable, even

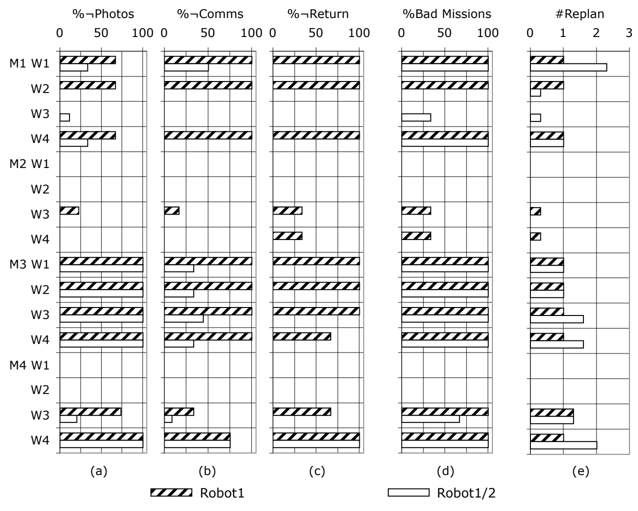

Figure 6: Results for mutation 1-589

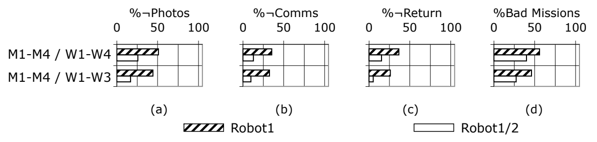

Figure 7: Impact of planner redundancy (with injected faults): *This figure presents overall results achieved for all 28 mutations, both with and without the heavily-constrained world W4.*

after recovery. This underlines the importance of plan analysis procedures to attempt to detect errors in plans *before* they are executed.

Conclusion

We presented in this paper a component, based on diversification, that tolerates software development faults in planners. 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). We also described its implementation in an existing robot architecture, using sequential planning associated with the first three error detection mechanisms.

To evaluate the performance overhead and the efficacy of our component, we introduced a validation framework using simulation of the robot hardware and fault injection by mutation of the planner declarative model. Previous experiments showed that the proposed component does not severely degrade the system performance in the chosen activities. In this article, we discussed example behaviors of the simulated robot in the presence of injected faults. In particular, several experiments demonstrated the necessity of integrated testing and numerous missions and environments to validate models supposed to take decisions in open environments. We also presented overall results that show that thie proposed component usefully improves the system behavior in the presence of model faults.

Several directions are open for future work. First, implementation of the plan analyzer detection mechanisms is central to allow better goal success levels to be achieved in the presence of faults, since it should increase error detection coverage and provide lower latency. The state of the art in planning already proposes several examples for off-line plan validation, but application for on-line validation still requires some work. Implementation of the concurrent planning policy and comparison with the sequential planning policy are also of interest. Moreover, note that the proposed mechanisms focus principally on *reliability* of the system: its efficacy in goal accomplishment. Other fault tolerance mechanisms are also needed to guarantee *safety* of the system and its direct environment.

References

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

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

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

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.; and Taylor, W. 2000. Remote Agent Experiment DS1 Technology Validation Report. Ames Research Center and JPL.

Brat, G.; Denney, E.; Giannakopoulou, D.; Frank, J.; and Jonsson, A. 2006. Verification of Autonomous Systems for Space Applications. In *IEEE Aerospace Conference 2006*.

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

Chen, I. R. 1997. Effects of Parallel Planning on System Reliability of Real-Time Expert Systems. *IEEE Transactions on Reliability* 46(1):81–87.

Crouzet, Y.; Waeselynck, H.; Lussier, B.; and Powell, D. 2006. The SESAME Experience: from Assembly Languages to Declarative Models. In *Proceedings of the 2nd Workshop on Mutation Analysis (Mutation'2006)*.

Daran, M., and Thévenod-Fosse, P. 1996. 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*.

Goldberg, A.; Havelund, K.; and McGann, C. 2006. Verification of Autonomous Systems for Space Applications. In *IEEE Aerospace Conference 2006*.

Havelund, K.; Lowry, M.; and Penix, J. 2001. Formal Analysis of a Space Craft Controller using SPIN. *IEEE Transactions on Software Engineering* 27(8):749–765.

Howey, R.; Long, D.; and Fox, M. 2004. VAL: Automatic Plan Validation, Continuous Effects and Mixed Initiative Planning using PDDL. In *16th IEEE International Conference on Tools with Artificial Intelligence*.

Joyeux, S.; Lampe, A.; Alami, R.; and Lacroix, S. 2005. Simulation in the LAAS Architecture. In *Proceedings of Principles and Practice of Software Development in Robotics (SDIR2005), ICRA workshop*.

Khatib, L.; Muscettola, N.; and Havelund, K. 2001. Mapping Temporal Planning Constraints into Timed Automata. In *Proceedings of the 8th International Symposium on Temporal Representation and Reasoning (TIME'01)*, 21–27.

Kim, K. H., and Welch, H. O. 1989. 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.

Lemai, S., and Ingrand, F. 2004. Interleaving Temporal Planning and Execution in Robotics Domains. In *Proceedings of AAAI-04*, 617–622.

Lussier, B.; Gallien, M.; Guiochet, J.; Ingrand, F.; Killijian, M. O.; and Powell, D. 2007. Fault Tolerant Planning in Critical Robots. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2007) (to appear)*.

Lussier, B. 2007. *Fault Tolerance in Autonomous Systems*. Ph.D. Dissertation, Institut National Polytechnique de Toulouse. in French.

Penix, J.; Pecheur, C.; and Havelund, K. 1998. "Using Model Checking to Validate AI Planner Domain Models. In *Proceedings of the 23rd Annual Software Engineering Workshop*.

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

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