# Robust Execution Monitoring for Navigation Plans

**Joaquín L. Fernández, Reid G. Simmons**
(joaquin@cs.cmu.edu, reids@cs.cmu.edu)
School of Computer Science/Robotics Institute
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh PA 15214

## Abstract

*This paper presents a general approach to robust execution monitoring. The goal is to provide coverage for many types of unexpected and unanticipated situations, while at the same time enabling the robot to quickly detect, and react to, specific contingencies. The approach uses a hierarchy of monitors, structured in layers of increasing specificity. We present the general approach, and show its application in the domain of indoor mobile robot navigation.*

## Introduction

Mobile robots operating in the real world need very reliable navigation capabilities to operate autonomously for long periods of time. Because it is almost impossible for the programmer to predict all the circumstances that might be encountered, a mechanism to handle the unexpected is required. We have developed an exception detection and recovery architecture for this purpose. The architecture uses hierarchically structured monitors. The upper layers of monitors are more general: They detect large classes of exceptions, however, they tend to be slow to react. The lower, more specific monitors use detailed knowledge of the domain to quickly detect task-specific situations. The basic philosophy is to use the general monitors to ensure coverage and then, based on experience, to add more specific monitors to detect the exceptions faster and get more information about the problem that causes the exceptions (for use in recovery).

For this paper, we take the role of execution monitors to be to detect *exceptions*. An exception is defined as a significant difference between the observed state-of-the-world and the *expectation* with respect to the *nominal* situation. The idea is that such exceptions will usually be associated with situations where the robot is stuck or is performing poorly.

We can represent the set of all the robot's states as in Fig. 1 where the *exception space* is the set of all the possible exception states and the *nominal space* (dark grey area) is the set of all the nominal states. The exceptions (represented as empty circles) are known in advance (for example, we can expect that sometimes the robot can find a blocked corridor). There are also exceptions that we do not anticipate and therefore we only know about them when the robot gets stuck in that situation. These exceptions are represented with black ellipses; the nominal situations are represented with empty squares. The goal, then, is to create execution monitors that cover the exception space as completely as possible, while still providing rapid detection of specific exceptional situations.

The difficulty is that there may be a large number of possible exceptions, and many of them may not even be anticipated by the system developer. To address this, we set up monitors to detect the *symptoms* of exceptions, where a symptom is a task-specific manifestation of poor performance. Depending on the metrics used to measure performance, we can set up different monitors to look for different symptoms. In navigation, for example, one metric of performance is the time it takes the robot to achieve its goal. Another measure is the distance traveled by the robot. In the latter case, a symptom will be that the robot is moving too slowly, or not moving at all.

Every monitor is associated with a symptom or a set of symptoms; These symptoms will cover a set of known exceptions, and may also cover a set of unanticipated exceptions. In Fig. 1 we represent the sets M0, M1, ... as the set of states detected for monitors m0, m1, .... Sometimes a symptom can also characterize a nominal situation because the nominal and exception states are too close and the symptom doesn't correctly split the exceptional and nominal space. Monitors m0, m2, m4 and m7 show that situation. Finding good symptoms that clearly split the exception and nominal states is not easy (especially since the definition of "poor performance" is somewhat subjective).

The set of monitors can be organized into a tree structure, from the more general to the more specific, where "more general" monitors cover a larger area of the exception space. For example, the tree in Fig. 2 represents the monitors corresponding to Fig. 1. General monitors are interesting because they detect many exceptions, but their disadvantages are that they may include more nominal situations, they usually take longer

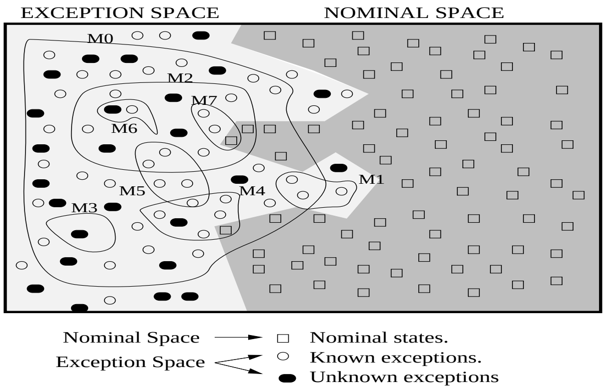


Figure 1: Exception space and subsets delimited by symptoms.

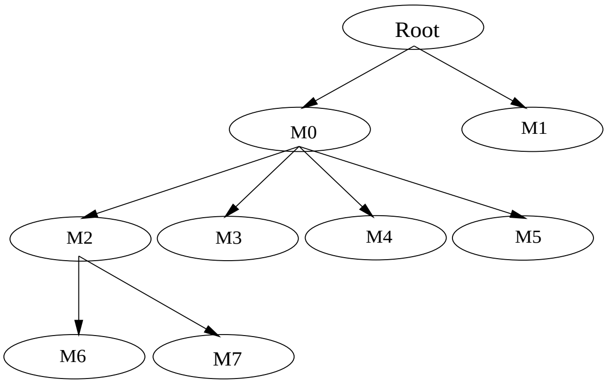


Figure 2: Monitor structure corresponding to Fig. 1.

to detect exceptions, and they do not provide much information about what the underlying problem was, for use in recovery. For example, a widely applicable type of general monitor is one that uses time as a symptom. The monitor is set up to fire when the time spent on the task is larger than the time the task needs in the worst case. This is a very general monitor since it detects almost all exceptions for all the tasks. However, if the problem occurs at the beginning of the task, the time from the problem's occurrence until the monitor detects the exception could be quite high. In addition, when the monitor fires one knows only that the task has not been achieved; The monitor provides no other information that might help in diagnosing the situation.

On the other hand, the monitors at the leaves of the tree provide more information about the type of exception. Besides getting more information about the exception, the specific monitors detect the situation faster. For these reasons, the more specific monitors are given preference if they fire at the same time as a general monitor. As an example, we use a specific monitor to detect the situation when the robot keeps spinning in the same position. That monitor counts the number of times that the robot rotates in the same position. This is a very specific monitor since it only detects that situation, however it can detect the situation faster and once the monitor is fired, we know exactly what the problem is so we can set up a special recovery algorithm for this situation.

While the general monitors can often be defined based on the task description and an *a priori* understanding of the environment, the need for the more specific monitors is often not known until one runs the robot for a while. For that reason, we start with the general monitors and add new and more specific monitors in order to detect the most common exceptions during the robot's life. While, eventually, we hope to have the robot learn the specific monitors autonomously, for now we hand-code the monitors and recovery procedures.

## Previous Work

Solutions to deal with errors depend strongly on the architecture used in the navigation system. For instance, error recovery is not used in pure reactive systems [1] because the system basically reacts to events. In the context of robot navigation, several architectures deal with exceptions (errors) in different ways [4] [10]. Monitoring and recovery processes are responsible for verifying that a robot is correctly executing its tasks, detecting when is not and handling exceptions.

In some architectures governed by planning [3] [9] the monitoring and recovery processes are used to check the correct completion of each step of the plan. Our monitoring and recovery architecture is independent of the navigation architecture. Even in reactive systems, there are always goals for the robot to accomplish, therefore we can set up monitors to evaluate how the robot is accomplishing the goal and check for some exceptions. The symptoms are dependent on the architecture since different architectures have to deal with different problems. Like other architectures, the architecture used in our research combines reactive and deliberative behaviors.

The approach described here has roots in earlier work with a previous navigation system [5]. The main differences are that this approach is based on using a hierarchy of monitors to ensure coverage of the exception space, and focuses on symptoms (poor performance) that are observable from task execution data.

## Xavier

Xavier, the robot used in these experiments, is designed with a layered architecture, consisting of task scheduling, path planning, navigation, and obstacle avoidance components [7]. This paper is mainly concerned with monitoring at the path-planning and navigation layers. Path planning uses a decision-theoretic generate, evaluate and refine strategy that is based on ideas from sensitivity analysis [2]. It tries to find paths that have the highest expected utility of travel, where utility is inversely proportional to travel time, and the planner takes into account the probability of the robot missing openings and the probability of finding blockages in the environment.

Navigation is performed using Partially Observable Markov Decision Process (POMDP) models [8]. The navigation system creates a POMDP model from a topological map of the building, augmented with approximate metric information. It takes the route produced by the path planner and creates a *policy* indicating how the robot should travel from each location. It then uses odometry and senor readings (sonar and laser) to update the probability distribution of the robot's position, and picks the action that has the highest total probability mass.

System integration is performed using the Task Control Architecture (TCA), a general framework for coordinating planning, sensing and real-time control [6]. TCA provides control constructs for inter-process communication, task decomposition, task synchronization, resource management, monitoring and exception handling. One of its novel features is that monitors and exception handlers can be incrementally added to the system, without needing to modify existing code [5]. This provides support for the "structured control" design approach, where systems are developed by first implementing behaviors that work in the nominal situation, and then incrementally adding on reactive behaviors that handle exceptions [6].

Although this POMDP-based navigation system is fairly reliable (achieving its goals about 95% of the time), there are still situations where it fails. In particular, if corridors are blocked, either temporarily (due to closed doors or people crowded around the robot) or permanently (due to structural changes in the environment), the POMDP-based navigation scheme will often loop forever. Thus, the system still needs explicit execution monitors to improve overall reliability.

## Hierarchical Monitors for Navigation

The monitors we developed for Xavier are structured as shown in Fig. 3. The monitors are implemented as an independent process that uses TCA control constructs to coordinate with the executing navigation tasks. In particular, we use the TCA "wiretap" mechanism [6] to determine when a new navigation task has started/ended, which tells the monitor process when to start/stop the set of monitors (except for the battery monitor, which is always running).

In the next sections, we describe the different symptoms used in the monitors, from the most general to the most specific. Each monitor uses different information (distance traveled, time, etc.) to produce a series of *tokens* that are used in the exception identification and recovery process (Section ).

### Symptom: Too Much Time

The most general monitor detects when the task has taken too long to complete. In fact, this *timer monitor* can be used for most types of tasks, since the only thing it needs to know about the task is the maximum estimated time for task completion. We first calculate the time that the robot needs to do this task, then add some exception interval determined by **time_margin** (% over the estimated time) and set a monitor to be fired at the end of this interval. In our case, travel time is estimated based on statistics acquired from some 2000 runs that Xavier has made over the past two years.

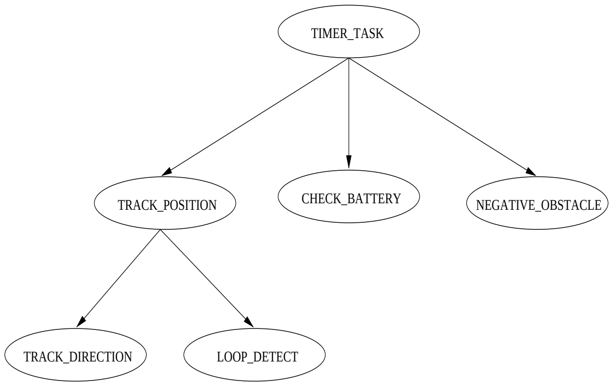


Figure 3: Xavier exception monitor architecture

The advantage of this monitor is that it has very wide coverage: Most exceptional situations result in long task achievement times. The disadvantage is that, even in normal situations, the variance is usually high (due to crowded environment, robot moving too slowly, etc.). In particular, to avoid excessive false positives, the monitor must be fired when the time spent on this task is greater than the time the task needs in the worst case. Also, the monitor may fire long after the problem actually arises, for instance, if the problem occurs near the beginning of a long task. For such reasons, we augment the timer monitor with additional, more specific, monitors.

### Symptom: Positional Error

Given that the robot is trying to follow a planned path, a class of exceptions can be detected by calculating the robot's expected position (assuming it travels at a constant speed) and comparing it with the current position (estimated by the POMDP navigation system). Situations that give rise to this symptom include where the corridor is crowded, and so the robot must slow down, where the robot misses a turn, and must circle back, and where a passage is blocked.

The positional error monitor periodically (once per second) updates the expected position (S) as if the robot was moving without any problem. It is also necessary to know the real position of the robot at the same time. This is complicated by the fact that the POMDP navigation system never knows the robot's position exactly. Instead, we use the probability distribution $P(s_i)$ over the Markov states that are mapped into different

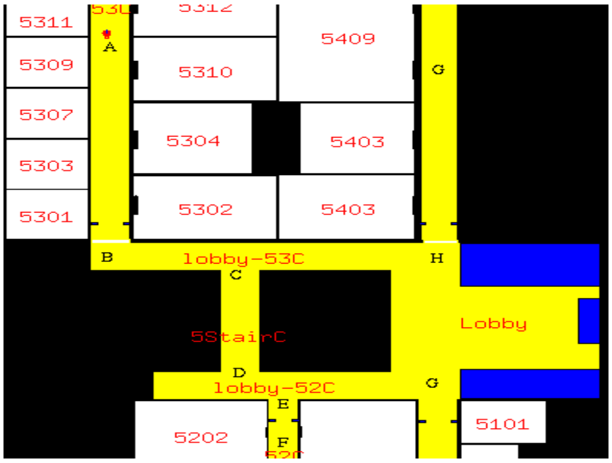


Figure 4: Map of the fifth floor in Wean Hall, CMU

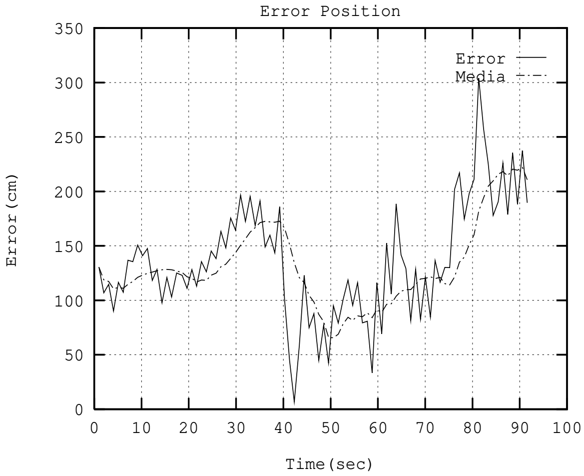


Figure 5: Position error while navigating from A to F

positions in the map $S_i$. Using both the expected position and the probability distribution, we can define the positional error as:

$$error = \sum_{i}^{n} \frac{P_i}{\sum_j^n P_j} \times d(S_i, S) \quad (1)$$

where $d(S_i, S)$is the Manhattan distance between $S_i$ and S, and n is the number of nodes in the probability distribution. [1]

Fig. 5 represents the error while the robot navigates from position A to F in the fifth floor of Wean Hall at Carnegie Mellon University (Fig. 4) using the path A-B-C-D-E-F. The solid line represents the measured error and the dashed represents the average error using a window of ten samples.

There are several factors that contribute to the noisiness of the graph. First, is the variability of the environment itself. Is almost impossible to predict the people that the robot is going to find in the way and therefore the time it is going to spend avoiding them. Second, the frequency of the positional error monitor is double the frequency of the update of the probability distribution, so for every two consecutive error samples, the distribution probability is the same but the expected position is changed according with the average speed. That is why the graph has a peak of every two samples.

Third, a characteristic of the POMDP navigation system is that the probability distribution tends to spread out when the environment is uniform, and then becomes sharp again when the robot observes distinctive features (such as corners and corridor intersections). While this is desirable behavior for navigation, for our purpose it produces undesirable fluctuations in the positional error measure.

[1] Instead of selecting all points in the distribution probability, only the points with probability higher than a low threshold are selected. That is the reason why in Equation 1 the division by $\sum P_j$ appears.

The consequence of all these problems is that the error is always accumulating as the robot is traveling. That means that we can not use the absolute error alone as a symptom of exceptional situations. However, what we want to know is when the robot is not making any progress at all, or when progress is really small.

It is possible to detect these situations using the derivative of the error rather than the error itself. We check for situations when the robot has made only small progress during some constant time **error_interval**. This means that the position monitoring system reports a possible error when it detects an increase of the average positional error greater than **max_error** during **error_interval** seconds. To select **error_interval** there is a trade off between fast detection of a blockage situation and mistakenly detecting blockage situations. The value was selected empirically to 20 seconds and the **max_error** a little lower than **max_error**×**average_speed**. Using the error in this way, even though the robot is slow or there are some sharp jumps in the error, the system does not detect an exception unless it almost stops during **error_interval**. Sometimes, the expected robot position reaches the goal location when the robot is actually still far from the goal. In this case, once the expected position reaches the end, the monitor merely looks for a constant average error (meaning no progress) during **error_interval**.

Figure 6 represents a typical example when the robot finds a blocked corridor. The change in positional error is approximately the average robot speed and the system detects an exception at time 73. At this point, the system plans a new path (see Section ), and the robot successfully continues to the goal.

### Symptom: Excessive Turning

Another typical symptom of possible problems is when the robot changes direction by 180 degrees. This sometimes occurs when the robot is confused about its position, but more often occurs when the robot has to turn

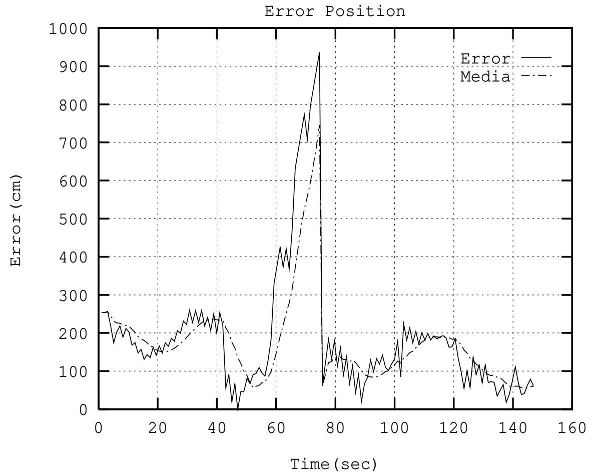


Figure 6: Error while navigating A-B-C-D-E-F with C-D blocked and replanning through C-H-G-E-F

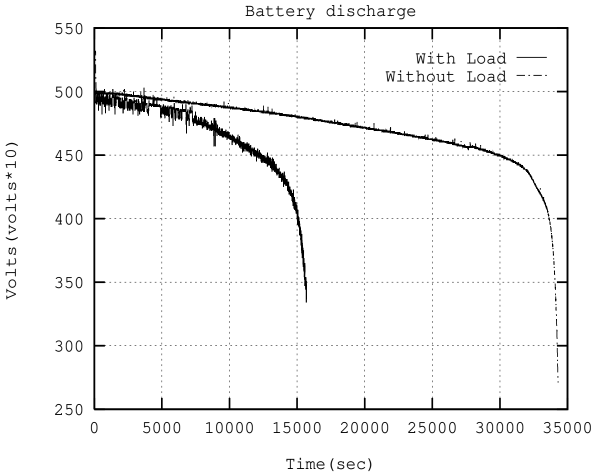


Figure 7: Battery level versus time with new batteries. The data were taken from two situations: robot moving all the time (with load) and with the robot still (without load)

in a T-shaped intersection (such as B-C-D-F in Fig. 4), but for some reason it misses turn (e.g., people may be blocking the intersection). In such situations, the robot travels forward until its POMDP models indicate that it has gone too far, and then it turns around and heads back. Under this circumstance, it is possible for the robot to keep going back and forth indefinitely unless it sees an opening.

Since it is not uncommon for the robot to miss an intersection once in a while (due to sensor noise), we monitor for this situation by checking for a series of 180 degree turns. We select the maximum number of times that the robot can miss the intersection (**max_180turns_allowed**) to be 3. While such situations can also be detected using the positional error monitor, most of the time this monitor gives us a faster detection and also better information about what is going on.

## Symptom: Spinning in Place

This is one of the most specific monitors we have developed so far. While it is not very common, sometimes the robot gets into a situation where it keeps spinning in one place indefinitely (we suspect this is due to a weird interaction between certain configurations of obstacles and the obstacle avoidance algorithm).

We set up a monitor to detect that situation using as symptom the number of turns that the robot does in the same position. This is a very specific monitor since, as far as we know, its symptom correlates with exactly one cause. This situation is also detected by the positional error monitor, but this monitor can detect the exception faster and, once the monitor is fired, we know exactly what the problem is and we can execute a special recovery algorithm for this situation.

## Symptom: Low Battery Level

We also implemented a monitor to avoid the situation in which the robot runs out of power in the middle of a task. The goal of this monitor is send the robot back to the nearest recharge station (so far is only the lab) before it will be in a situation where the battery level is too low to return.

Since we want the monitor to fire when the robot still has enough battery charge to return to base, we must characterize the performance of the batteries over time. At full charge, the batteries supply 56 volts; Empirically we know that the robot starts having problems when the voltage drops below 32 volts. We ran Xavier and collected data for two situations (Fig. 7): (a) the robot was moving until it ran out of power; (b) the robot was still all the time (no load) until it ran out of power (we also collected data with old batteries – the curves have the same shapes, but the drop off times are much sooner).

From Fig. 7, we can see that is important take into account the movement of the robot in order to calculate the expected battery level. Therefore, to calculate the battery level at the end of a task, it is necessary to estimate the time that the robot needs to do the task and which portion of this time it is going to be moving. The monitor uses information from the planner to calculate the **time_running** and **time_idle** estimated that the robot needs to come back to the lab. Using these estimates, the actual (measured) battery level , and the information from the graphs shown in Fig. 7, the system estimates the battery level **end_level** needed to finish the task and return to the lab. To avoid excessive computation, the monitor periodically checks whether the battery level is below a constant level **batt_level_check**, and only then does it check if

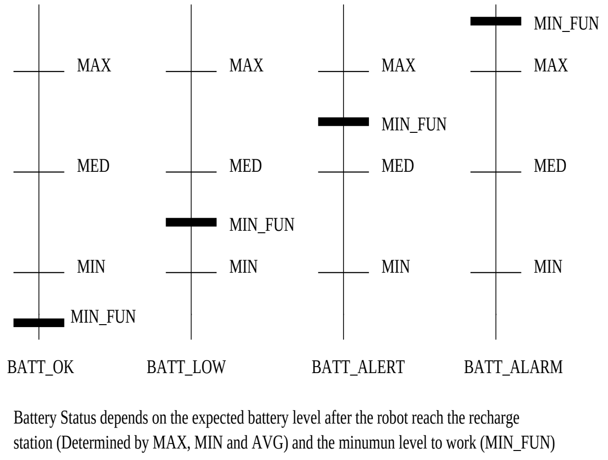


Figure 8: Battery status.

the charge is enough to finish the current task and come back to recharge.

The **time_running**, **time_idle** and therefore the **end_level** are actually intervals with a maximum, minimum and average level instead of fixed values. This way, depending on the **end_level**, the battery status is classified as one of the four possible status according with the Fig. 8.

## Exception Identification and Recovery

While our work to date has mainly focused on robust execution monitoring, we have begun to use the information produced by the monitors to identify possible causes of the exceptions and to have the robot attempt to recover autonomously.

Using all the information from the different monitors, the system attempts to determine what the problem is, and how to recover from it. When the exception identification procedure gets a new token from one of the monitors described in the last section, it registers this token and decides if there is a possible error, based on the information from this new token. In case multiple monitors send tokens simultaneously, preference is give to monitors lower in the hierarchy, since they provide more information about the possible cause of the problem.

While the timer monitor has wide coverage of the exception space, its firing gives little indication about what the actual problem might be (in fact, this is precisely because it has wide coverage – almost anything that goes wrong can trigger this monitor). In the absence of any other information, the best the robot can do is to stop its task and report back failure to a human. On the other hand, the "spin in place" monitor leads to a very specific recovery action – merely stopping the robot and turning in the direction of the largest free space (based on sonar readings) is usually sufficient to get Xavier back on track.

Similarly, the "battery level" symptom is identified with a particular problem. The recovery action, however, depends on the severity of the problem. This is indicated by the type of token produced by the battery monitor:

**BATT_OK:** Nothing is done.

**BATT_LOW:** Finish the actual goal, go to recharge.

**BATT_ALERT:** Stop the robot, reset and replan to go to recharge.

**BATT_ALARM:** Stop the robot, reset and replan to go to recharge.

For all other monitors, the problem is presumed to be a blockage of some sort (this is admittedly a simplistic approach, and we are working to incorporate more sophisticated diagnoses of the possible causes for the symptoms, for instance, by looking more closely to see if there is really a blockage, or by relocalizing if the probability distribution is too spread out). To recover from a blocked path, the system must first determine where the blockage may have occurred. To do this, the exception recovery system gets a copy of the path every time a new navigation task is started. During the navigation, it keeps a record of which map arcs are traversed, based on the changes to the Markov probability distribution.

When the monitor triggers, the system finds the last arc on the path that has been traversed, and assumes that the next arc on the planned path is the one blocked. It then tells to planner to increase the probability that that arc was blocked, and tries to reachieve the navigation goal (assuming that the planner will produce a different path). Even if the wrong arc is blamed, the robot will eventually keep replanning until it finds a traversable path, since the exception handler only modifies the traversal probability, but never actually eliminates any arc from consideration.

## Summary and Conclusions

We have presented a general approach to execution monitoring that uses hierarchically structured monitors to detect *exceptions* between the expected and observed states of the world. The top-most, more general monitors have wide coverage, but tend to yield little information about the cause of the exception. More specific monitors are used to quickly detect and react to particular classes of exceptional situations.

We presented the use of this approach in the context of indoor mobile robot navigation. While we do not have complete coverage of the exception space (for instance, no on-board monitor can detect when all the robot's computers crash simultaneously), the monitors we have developed cover a very wide range of the exceptions that commonly occur in indoor navigation. Some of these monitors need little domain information, and were very easy to encode. Most were very task-specific, and needed to be tuned for our particular robot and the environment in which it normally operates (for instance, battery characteristics, or average speed). While we would like to learn such monitors automatically, our experience indicates that it would be a difficult task, in general. For now, we continue to experiment with the set of hand-coded monitors we have developed, and are working on adding more sophisticated recovery strategies.

In general, however, the approach of structuring the monitors hierarchically has been a big benefit in thinking about ways to detect both general and specific exceptions. We have confidence that the approach will scale well, and will be applicable to other dynamic domains.


## Acknowledgments

We would like to thank all the people that have influenced this work. In particular to Greg Whelan, for helping with the TCA tools, Greg Armstrong, and all the other people from the Robotics Learning Lab. We also like to thank the "Provigo Foundation" for funding Joaquin's visit to the Robotics Learning Lab.


## References


[1] R. A. Brooks. A robust layered control system for a mobile robot. *IEEE Transactions on Robotics and Automation*, RA-2(1):14–23, 1986.

[2] S. Koenig, R. Goodwin, and R.G. Simmons. Robot navigation with Markov models: A framework for path planning and learning with limited computational resources. In L. Dorst, M. van Lambalgen, and R. Voorbraak, editors, *Reasoning with Uncertainty in Robotics*, volume 1093 of *Lecture Notes in Artificial Intelligence*, pages 322–337. Springer, 1996.

[3] Fabrice R. Noreils. Integrating Error Recovery in a Mobile Robot Control System. In *IEEE Int. Conf. Robot. and Automat.*, pages 396–401, 1990.

[4] Fabrice R. Noreils and G. Chatila. Plan Execution Monitoring and Control Architecture for Mobile Robots. *IEEE transactions on robotics and automation*, 2:396–401, April 1995.

[5] Reid Simmons. Becoming increasingly reliable. In *Proc. of 2nd Intl. Conference on Artificial Intelligence Planning Systems*, Chicago, IL, June 1994.

[6] Reid Simmons. Structured control for autonomous robots. *IEEE Transactions on Robotics and Automation*, 10(1):34–43, February 1994.

[7] Reid Simmons, Rich Goodwin, Karen Zita Haigh, Sven Koenig, and Joseph O'Sullivan. A layered architecture for office delivery robots. In W. Lewis Johnson, editor, *Proceedings of Autonomous Agents '97*, pages 245–252, Marina del Rey, CA, February 1997. ACM Press.

[8] Reid Simmons and Sven Koenig. Probabilistic Navigation in Partially Observable Environments. In *Proceedings of the International Joint Conferende on Artificial Intelligence (IJCAI)*, pages 1080–1087, 1995.

[9] Elizabeth Robertson Stuck. *Detecting and Diagnosing Mistakes in Inexact Vision-based Navegation.* PhD thesis, University of Minnesota, November 1992.

[10] Elizabeth Robertson Stuck. Detecting and Diagnosing Mistakes. In *Proceedings of the IEEE/RSJ International conference on Intelligent Robots and Systems*, August 1995.