Fault Tolerance of a Reconfigurable Autonomous Goal-Based Robotic Control System

Julia M. B. Braman, David A. Wagner, and Richard M. Murray

Abstract—Fault tolerance is essential for the success of autonomous robotic systems. A control architecture, called Mission Data System (MDS), developed at the Jet Propulsion Laboratory (JPL), takes a goal-based control approach. The ability of the control system to reconfigure by re-elaborating failed goals suggests an increased degree of tolerance to certain failures. In this paper, an autonomous robotic task that involves some decision-making is simulated using MDS as the control architecture. Sensor failures and degradations are introduced during the task, and completion and success rates are recorded and analyzed. Results show that the ability of MDS to re-elaborate goals in response to sensor failures improves the success and completion rates of the robotic task.

I. INTRODUCTION

Autonomous robotic missions by nature have complex control systems. In general, the necessary fault detection, isolation and recovery software for these systems is cumbersome and added on as failure cases are encountered in simulation. There is a need for a systematic way to incorporate fault tolerance in autonomous robotic control systems. One way to accomplish this could be to create a flexible control system that can reconfigure itself in the presence of failures. However, the added complexity of the reconfigurability of a system could reduce the system’s effective fault tolerance.

Mission Data System (MDS) is a software control architecture that was developed at the Jet Propulsion Laboratory (JPL) [1]. It is based on a systems engineering concept called State Analysis [2]. Systems that use MDS are controlled by goals instead of by sequences of commands. Goals express the intent of a string of commands by constraining states of the system in time. By encoding the intent of the robot’s actions in the control system, MDS has naturally allowed more fault response options to be explored by the control system without human intervention [3].

A great deal of work to date has focused on issues related to detecting and recovering from sensor failures in the control of autonomous systems [4]. Multiple model-based methods use several Kalman filters, one for each nominal or failure mode, in parallel or mutually interacting, to detect and identify failures [5]. Particle filters, which can represent discrete and continuous states simultaneously, have also been used to detect faults by representing several modes of operation (nominal and failure), and then probabilistically determining

the operational mode using sensor measurements [6]. Hybrid estimation with an “unknown mode” capability has been developed to detect gradual or partial failures [7]. Other methods in sensor fusion and fault detection include generating probability functions using Bayesian theory, generating belief functions using the Dempster-Shafer theory, or using weight of conflict, voting, or trust methods [8]. These methods all attempt to autonomously reconfigure the control system by changing the estimation scheme or parameters.

In practice, many of these methods are cumbersome. Most methods suffer from complexity issues; complex autonomous systems with many sensors and therefore many failure states would need a computationally costly particle filter, or a large number of Kalman filters in the multiple model-based methods. Most of the multiple model and sensor fusion methods only detect hard, or total, sensor failures reliably and have trouble detecting and correcting for slow degradation of sensor ability. Some of the sensor fusion methods have reduced failure detection ability when in uncertain or unknown environments. Multiple sensor failure diagnosis is not handled well by most of the current methods; more Kalman filters or failure states must be added to increase the complexity of the multiple model or particle filter approaches, respectively, and sensor fusion methods lose capability when more than one fault occurs due to their community approach to finding faulty sensors.

Several fault tolerant control architectures for autonomous systems have been developed in which the control effort is layered to deal with faults on different levels, including low levels of hardware control and high levels of supervisory control [9], [10]. Another fault-tolerant control architecture, ALLIANCE, is a behavior-based control system for multi-robot cooperative tasks [11]. In ALLIANCE, the distributed control system re-allocates tasks between robots in response to failures.

Although many fault tolerant control systems boast reconfigurability, few actually change the commands given to the system. One system uses adaptive neural/fuzzy control to reconfigure the control system in the presence of detected faults [12], and another reconfigures both the control system design and the inputs to the control system [13], though neither adjusts the intent of the commands in response to failures.

In this paper, MDS is used as the goal-based control architecture for a representative robotic task and is evaluated for its fault tolerance qualities. This work is the first to specifically study the role of goal re-elaboration in the fault tolerance of a complex system. It is demonstrated that
the capability of MDS to handle re-elaboration of its goal network indeed improves the fault tolerance of the system.

This paper is organized as follows. Section II describes the main features of MDS. Section III discusses the design of the robotic task and simulation. Section IV presents the results of introducing simulated sensor failures to the robotic system while it is attempting to complete its task, and section V concludes the paper and discusses the utility of MDS for fault tolerant control.

II. MISSION DATA SYSTEM OVERVIEW

A. State Analysis

State Analysis is a systems engineering methodology that focuses on a state-based approach to the design of a system [2], [14]. In State Analysis, the control system and the system under control are considered separately. Models of the system under control are present in the control system and are used for such things as the estimation of state variables, control of the system, and projection. State variables are representations of states or properties of the system that are to be controlled or that affect a controlled state. Examples of state variables could include the position of a robot, the temperature of the environment, the health of a sensor, and the position of a switch.

State Analysis is applied in the following fashion. First, the state variables of the system under control are identified. Next, a model of the system under control is developed, consisting of three types of models: measurement, command, and estimation. Measurement models describe the effect of the physical state of the system under control on the measurements, command models describe the effect of actuator commands on physical state of the system under control, and estimation models describe how measurement evidence and other state variables affect the state variable estimation of the physical state. Next, the controllers and estimators are designed using the command, measurement, and estimation models. Finally, goals and goal elaborations are created.

Goals are specific statements of intent used to control a system by constraining a state variable in time. More complex parent goals are designed to be elaborated into child goals. The simplest goals lead directly to commands. Which goals are elaborated from a parent goal are based on both the intent and type of goal, and on several rules, listed in [2].

B. Mission Data System

A core concept of State Analysis is that the language used to design the control system should be nearly the same as the language used to implement the control system. Therefore, the software architecture, Mission Data System, is closely related to the systems engineering theory described in the previous section. Figure 1 gives a visual representation of MDS.

Data structures called state variables are central to MDS [15]. These state variables directly correspond to the state variables from State Analysis. A state variable structure can contain much information; for example, a position state variable for a robot in the plane could contain the robot’s (x, y) position, its velocity in component form, and uncertainty values for each piece of information. Each state variable has a unique estimator, and if necessary, a controller. State function histories are created for each, and they store the past values of the state variable. These can also store future projections of the state variable’s value based on the current goal net. Constraint structures can be created that affect some or all of a state variable’s information. For example, a constraint could be placed on the velocity of the position state variable used in the previous example, but could leave the position or uncertainties unconstrained.

Goal networks replace command sequences as the control input to the system. Goals are created with different levels of complexity. The simplest goal directly constrains a single state variable in a way that a single type of control action or estimation occurs. These goals are achieved by the estimator or controller of the state variable that is constrained. More complex goals may cause constraints to be elaborated on many state variables, or constrain one state variable so that many actions need to occur. These goals declare the intent of the control action and must have an associated elaboration class. This elaboration class instructs the elaborator in MDS to add certain goals to the goal network in support of the parent goal. Elaboration can be layered; a goal can elaborate a goal that elaborates several others, and so on.

Elaboration allows MDS more flexibility to handle tasks with which traditional command sequence control architectures struggle. One example is fault tolerance. If there is physical redundancy in the system, if there are many ways to accomplish the same task, or if there are degraded modes of operation that are also acceptable for a task, re-elaboration of failed goals is an option. The elaboration class for a goal can include several pre-defined tactics. These tactics are simply different ways to accomplish the intent of the goal, and tactics may be logically chosen by the elaborator based on programmer-defined conditions. This capability allows for many common classes and combinations of faults to be accommodated automatically. It gives the control system some ability to reason about a failure situation and attempt to recover from it [3].

MDS is still under development at JPL and some large scale examples of its use are being studied. Current work
includes using a version of MDS to control the Deep Space Network Array, nominally and in the presence of failures that induce goal re-elaboration.

III. SIMULATION AND TASK DESIGN

The robotic simulation environment used in this example consists of three software packages. The autonomous robotic control system is implemented in MDS, and an open-source 3-D robotic simulation package called Gazebo is used to simulate the environment, the robot, and its sensors. An open-source server package called Player is used to interface between the hardware adapter in MDS and the simulated robot in Gazebo.

A. Task

The modeled autonomous robotic task is for a simulated Pioneer robot with several sensors to follow a path within a given uncertainty bound. The planned route consists two checkpoints, \( c_1 \) and \( c_2 \); after the first checkpoint, \( c_1 \), there are two possibilities for the location of \( c_2 \), \( p_1 \) and \( p_2 \). The first of these possibilities, \( p_1 \), lies down a path that has a somewhat tighter error bound and requires a higher standard of instrument health. The other possibility, \( p_2 \), lies down a second path that allows for a larger error bound and a somewhat degraded sensor capability.

The path is successfully navigated by the robot if the robot stays within the path boundaries as shown in Figure 2. The boundaries represent the error bounds allowed down each path. Completion of the task occurs when the robot navigates to and stops sufficiently near \( c_2 \) without breaching the boundary.

The second checkpoint, \( c_2 \) is first assigned to be \( p_1 \), but can be changed to be \( p_2 \) upon re-elaboration of the initial goal. This re-elaboration occurs if a goal constraining the sensor system health to the necessary high standard fails. The task could be compared to a Mars scientific mission in which there are two points of interest (\( p_1 \) and \( p_2 \)) and the first is more desirable but needs a lower uncertainty of the robot’s position to reach it. The mission is considered a success if the robot does not wander off path (where it could be damaged or get stuck), and the mission is completed if the robot reaches either point of interest.

B. System Design

1) Sensors: The Pioneer robot is outfitted with three sensors: differential GPS, Ladar, and Odometry (the collection of position, orientation, and velocity information deduced from wheel encoders). These three sensors are used to estimate the robot’s position, orientation, and velocity information. Several obstacles were placed outside the boundaries of the course to facilitate the use of the Ladar. The scan matching algorithm developed by Lu and Milios [16] was adapted for use in this simulation. The output of this algorithm is position and orientation, although only the orientation estimate is used. The Odometry measurements have increasing uncertainty and so are reset to the current position and orientation estimates at regular intervals.

2) State Variables: Seven state variables are needed to describe this system. First, the position and the orientation of the robot are separated into Position and Angle state variables. These state variables track cartesian and angular position and velocity, as well as the covariance matrices for the estimates. Second, the Ladar scan matching outputs are stored in the LadarScan state variable. Three state variables describe the health of the three sensors. Finally, the health of the overall robotic sensing system is described by the SystemHealth derived state variable [15].

3) Estimation: The position and orientation state variables are estimated using a multiple model-based method [17]. In order to make the estimation algorithm robust to changes in sensor availability and health, different Kalman filters were designed for each possible combination of sensors. This approach was chosen for its relative simplicity and ease of implementation. Position and orientation are modeled in similar ways. The values that are updated are the position (\( x,y \)) and orientation, \( \theta \), and the corresponding velocities (\( \dot{x}, \dot{y}, \dot{\theta} \)). The system equation used to predict state is

\[
\dot{x}_{kp} = A\hat{x}_{k-1} + Bu
\]

where \( A = \begin{bmatrix} 1 & \Delta t \\ 0 & 0 \end{bmatrix}, B = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \hat{x}_k = \begin{bmatrix} x \\ \dot{x} \end{bmatrix}, \begin{bmatrix} y \\ \dot{y} \end{bmatrix} \), or

\[
\begin{bmatrix} \theta \\ \dot{\theta} \end{bmatrix}, u \) is the respective command velocity and \( \Delta t \) is the time step. This model was chosen because the commands that Gazebo accepts for the Pioneer robot are the forward and angular velocities, not forces. The predicted estimate covariance equation is

\[
P_{kp} = AP_{k-1}A^T + Q
\]

where \( P_{kp} \) is the predicted covariance matrix for the estimate and \( Q \) is the covariance matrix for the process noise.

The discrete Kalman filter for the system continues with the update steps. The optimal Kalman gain is

\[
K = P_{kp}H^T(HP_{kp}H^T + R)^{-1}
\]

where \( H \) is the matrix that relates the sensor measurements to the state vector and \( R \) is covariance matrix of the sensor.
noise. The sensor health state variables help to pick the correct Kalman filter and also update the observation covariance matrix, $R$. Next, the state is updated using

$$\dot{x}_k = \dot{x}_{kp} + K(Z - \hat{H}_k)$$

(4)

where $Z$ is the vector of measurements. Finally, the estimate covariance matrix is updated using

$$P_k = (I - KH)P_{kp}$$

(5)

where $I$ is the identity matrix.

The three sensor health variables are estimated using a different process. In each sensor’s health estimator, the output of the sensor is converted to a measured position and velocity value. For the GPS and Ladar sensors which do not directly measure velocity, an average change in position over the previous time step is calculated. The position and/or orientation values from each sensor are then compared to the others. The velocity values are compared to the commanded velocity or angular velocity, if applicable. Each comparison produces a number that describes how close the values are to each other. Achieving the tightest bound produces the number 1; if the difference between the values exceeds all specified error bounds, a 4 is produced.

Each comparison value is then added to produce three sums; for example, the sums in the Odometry Health estimator would be $og + oc$, $og + ol$, $oc + ol$, where $og$ is the comparison value between the Odometry and GPS measurements, $oc$ is the comparison value between the Odometry and measurement and the commanded velocity, and $ol$ is the comparison value between the Odometry and Ladar measurements. The smallest of these three values then determines the health of the sensor as shown in Table I. Special emphasis, however, is given to the comparison between the measurement and the commanded velocity in steady state and when the command is non-zero because there is no possibility that the command has failed.

If the GPS or Odometry is failed, it is assumed that it will always be failed. Therefore, the estimation algorithm no longer runs for that sensor’s health state variable, and the comparisons between the failed sensor measurements and the other sensor measurements are no longer made. So, only one comparison value sum is present, and a slightly different formula is used to find the health values, as seen in Table II.

The System Health derived state variable is estimated using the health state variables of the three sensors. The system’s health is good when the GPS sensor (the most accurate sensor) is in good or fair health and one other sensor has some health. The system health is fair when the GPS is the only healthy sensor or if the GPS has poor health but the other two sensors are healthy. The system health is poor when at least one sensor isn’t failed, and is a failed system when all sensors are failed.

### C. Goal Design

The task’s goal net consists of several types of goals. First, goals ensuring system health and the knowledge of the sensor health states are added over the entire task period. Next, a goal called BeAtPosGoal, which constrains the position and orientation of the robot over some time period, is added to allow the robot to move to the position of the first checkpoint, $c_1$. Then, a macro goal called the DecisionGoal is added. This goal has several tactics. In the first tactic, an UncertGoal is used to constrain the system health to be good. Then, a BeAtPosGoal is added to constrain the robot to be at the checkpoint $c_2$, which is set for the position of $p_1$ at the end of the corridor.

If the initial UncertGoal fails, the second tactic of the DecisionGoal is elaborated. The second tactic constrains the system health to be fair or better and employs a new BeAtPosGoal to again constrain the robot to be at the second checkpoint, $c_2$. However, the position of $c_2$ is assigned to be $p_2$, which is at the end of the wider corridor. The third and final tactic of the DecisionGoal is elaborated when the UncertGoal in the second tactic fails at any time, or if the UncertGoal of the first tactic fails after the BeAtPosGoal has started to be achieved. The third tactic consists of a StopGoal, which nulls all previous velocity commands to protect the robot from breaching the boundaries. Figure 3 shows both the goal network and the three tactics of the DecisionGoal.

The BeAtPosGoal mentioned above elaborates to three goals: a maintenance goal on the position state variable;
BeAtAngGoal, which constrains the robot to be at a given orientation; and a GoToPosGoal, which is a transition goal that allows the position in the parent goal to be achieved by the given time.

IV. RESULTS

The robotic task was simulated in a nominal case and in several sensor failure and degradation cases. Each sensor was failed by intercepting the sensor measurement values supplied by Gazebo and setting them to some constant value. The sensor degradations were simulated in different ways: the GPS degradation only affected the orientation output, the Odometry degradation scaled all the position, orientation, and velocity information by half, and the Ladar degradation was simulated by only affecting a subset of the scan range points. Each of the failures and degradations were introduced at eight different time points during the task.

The nominal case performed exactly as expected, as seen in Figures 4 and 5. The health of the system and sensors remains high throughout the run, and the checkpoints are achieved in order, with checkpoint $c_2$ occurring at $p_1$, which has the narrower error bound. The Ladar and Odometry failure and degradation cases were similar to the nominal case in that each case was successful and the route to $c_2$ occurring at $p_1$ was completed in each case.

The GPS failure and degradation cases were more interesting. The cases fell in two major categories: one, in five runs, the GPS failure or degradation occurred before the robot starting turning toward $c_2$, and two, in the remaining three runs, the failure or degradation occurred after the robot starting moving toward $c_2$. All of the runs in the first category, represented in Figures 6 and 7, were successful and all attempted to reach $c_2$, which was now set to $p_2$ after the re-elaboration of the failed DecisionGoal. The only run that did not reach $c_2$ came to a stop just outside the acceptable error bound around the checkpoint. The runs in the second category, however, did not have a perfect success rate and had a zero completion rate. In these runs, since the robot was already en route to $c_2$, which was set to the position of $p_1$, a successful run upon degradation of the system health was for the robot to stop before traversing the route boundaries. All but one run was successful.

The GPS failure and degradation cases were re-run with the re-elaboration capability disabled. The system health was estimated, but no goal was constraining the system to have a certain health value. Therefore, only one tactic of the DecisionGoal was available, effectively removing the reconfigurability from the system. The overall success rate for all the runs dropped to only 75% from 94% and the completion rate dropped to 50% from 56%, as seen in Table III.

The completion rates of the re-elaborated runs with post-$c_1$ degradations and failures is 0% because of the third tactic of the DecisionGoal, the StopGoal. This is elaborated upon failure of the UncertGoal on the system health after the BeAtPosGoal for $c_2$ begins to be achieved and the effect of this goal is to stop the robot. It is apparent by looking at the completion and success rates of the static post-$c_1$
TABLE III
COMPARISON BETWEEN RE-ELABORATED (R) AND STATIC (S) GPS
DEGRADATION AND FAILURE RUNS WITH FAILURES BEFORE AND
AFTER THE FIRST CHECKPOINT

<table>
<thead>
<tr>
<th>Deg. Pre-Chkpt</th>
<th>R Success</th>
<th>R Complete</th>
<th>S Success</th>
<th>S Complete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deg. Post-Chkpt</td>
<td>100%</td>
<td>80%</td>
<td>100%</td>
<td>60%</td>
</tr>
<tr>
<td>Fail Pre-Chkpt</td>
<td>100%</td>
<td>0%</td>
<td>100%</td>
<td>67%</td>
</tr>
<tr>
<td>Fail Post-Chkpt</td>
<td>67%</td>
<td>0%</td>
<td>100%</td>
<td>67%</td>
</tr>
</tbody>
</table>

runs that the re-elaborated runs are conservative. Successful completions that may have occurred even with the degraded sensor measurements were stifled. However, pre-c1 failures and degradations clearly were benefitted by the ability to re-elaborate the goals.

Currently, there are some deficiencies in the re-elaboration process. The only re-elaboration case that did not succeed is an example. This failure occurred because after the Uncert-Goal failed, the executable goals on the position and orientation were not deleted, and they continued achieving until the new goal net was scheduled. That caused the robot to drive over the boundary. This could be solved by handling re-elaborations in a different way, such as deleting all children goals of a failed parent immediately and replacing them with unconstrained or maintenance goals until the new goal net can be scheduled. Scheduling of the re-elaborated goal net in this example while other executable goals were achieving took twenty to twenty-five time steps, which is long enough for the robot to get into trouble in some situations.

V. CONCLUSIONS
The ability of MDS to re-elaborate goals and effectively reconfigure the control system in the presence of sensor failures significantly increases the success rate of the simulated task. The Ladar and Odometry failure and degradation cases showed that the health variable design and estimation methods could successfully identify and remove the influence of a faulty sensor from the position and orientation estimation. The reconfigurability of the goal net suggests that MDS is better suited to the fault tolerant control of autonomous systems than less complex static control systems. The use of a goal-based control architecture such as MDS brings added complexity and structure to the design of the system and control inputs, but in goal re-elaboration, adds a relatively simple way to improve the fault tolerance of the system.

VI. ACKNOWLEDGEMENTS
The authors would like to gratefully acknowledge Kenny Meyer for his many efforts in enabling this work between Caltech and JPL; Michel Ingham, Robert Rasmussen, Daniel Dvorak, and the MDS team at JPL for feedback, suggestions, answered questions, and MDS and State Analysis instruction; and Jeremy Ma for supplying knowledge and code for the Lu and Milios scan matching algorithm.

REFERENCES