High-speed Obstacle Avoidance for Mobile Robots

by
J. Borenstein and Y. Koren

Department of Mechanical Engineering and Applied Mechanics
The University of Michigan, Ann Arbor

ABSTRACT

A new real-time obstacle avoidance approach for mobile robots has been developed and implemented. This approach permits the detection of unknown obstacles simultaneously with the steering of the mobile robot to avoid collisions and advancing toward the target. The novelty of this approach, entitled the Virtual Force Field, lies in the integration of two known concepts: Certainty Grids for obstacle representation, and Potential Fields for navigation. This combination is especially suitable for the accommodation of inaccurate sensor data (such as produced by ultrasonic sensors) as well as for sensor fusion, and enables continuous motion of the robot without stopping in front of obstacles.

Experimental results from a mobile robot running at a maximum speed of 0.78 m/sec demonstrate the power of the proposed algorithm.

1. Introduction

One widely used method for obstacle avoidance is based on edge detection. In this method, the algorithm determines the position of an obstacle's vertical edges and consequently attempts to steer the robot around those edges. This method was used in our own previous research (Borenstein, 1987; Borenstein and Koren, 1988), as well as in several other research projects, such as (Crowley, 1984; Weisbin et al. 1986).

A drawback of this method is its sensitivity to sensor accuracy. Unfortunately, ultrasonic sensors, which are commonly used in mobile robot applications, have many shortcomings that aggravate this drawback:

1. Poor directionality limits the accuracy in determining the spatial position of an edge to 10-50 cm, depending on the distance to the obstacle.

2. Frequent misreadings, caused by either ultrasonic noise from external sources or stray reflections from neighboring sensors, cannot always be filtered out.

3. Specular reflections occur when a smooth surface of an unfavorably oriented obstacle reflects incoming ultra-sound waves away from the sensor. In this case, the obstacle either is not detected or is "seen" as much smaller than it is in reality (since only part of the surface is detected).

Another disadvantage with an obstacle avoidance approach based on edge detection is the need for the robot to stop in front of an obstacle in order to obtain more accurate measurements.

2. The Certainty Grid for Obstacle Representation

The representation of obstacle in a grid model using certainty levels has been suggested by Elfes (1985), Moravec and Elfes (1985), and Moravec (1986). This representation is especially suited to the unified representation of data from different sensors (ultrasonic, vision, proximity, etc.) as well as the accommodation of inaccurate sensor data (such as measurements from ultrasonic sensors).

With the certainty grid world model, the robot's work area is represented by a two-dimensional array of square elements (denoted as cells). Each cell (i,j) contains a certainty value C(i,j) that indicates the measure of confidence that an obstacle exists within the cell area. The greater C(i,j), the greater the level of confidence that an obstacle occupies the cell.

With our approach, ultrasonic sensors are continuously sampled while the robot is moving. If an obstacle produces an echo (within the predefined maximum range limit of 2 m), the corresponding cell content C(i,j) is incremented. A solid, motionless obstacle will eventually cause a high count in its corresponding cells. Misreadings, on the other hand, occur randomly, and will not cause high counts in any particular cell. Thus, this method yields a more reliable obstacle representation in spite of the ultrasonic sensors' inaccuracies.

This work was sponsored (in part) by the Department of Energy Grant DE-FG02-86NE37969

0-8186-2012-9/89/0000/0382$01.00 © 1989 IEEE

3. The Virtual Force Field (VFF) Algorithm

The idea of obstacles conceptually exerting forces onto a mobile robot has been suggested by Khatib (1985). Krogh (1984) uses a similar concept which takes into consideration the robot's velocity in the vicinity of obstacles, and Thorpe (1985) uses the Potential Fields Method for off-line path planning. Krogh and Thorpe (1986) suggest a combined method for global and local path planning that uses Krogh's Generalized Potential Field (GPF) approach.

These methods, however, assume a known and prescribed world model of the obstacles. Furthermore, none of the above methods has been implemented on a mobile robot that uses real sensory data. The closest project to ours is that of Brooks (1986; 1987), who uses a Force Field method in an experimental robot equipped with a ring of 12 ultrasonic sensors. Brooks's implementation treats each ultrasonic range reading as a repulsive force vector. If the magnitude of the sum of the repulsive forces exceeds a certain threshold, the robot stops, turns into the direction of the resultant force vector, and moves on.

The combination of the Potential Field method with a Certainty Grid produces a powerful and robust control scheme for mobile robots. This approach has been denoted as the Virtual Force Field (VFF) method.

We have implemented the VFF method on our mobile robot, CARMEL. When CARMEL moves around, range readings are taken and projected into the Certainty Grid, as explained above. Simultaneously, the VFF algorithm scans a small square window of the grid. The size of the window is 33x33 cells (i.e., 3.30x3.30m) and its location is such that the robot is always at its center.

Each occupied cell inside the window applies a repulsive force to the robot, "pushing" the robot away from the cell. The magnitude of this force is proportional to the cell contents, C(i,j), and is inversely proportional to the square of the distance between the cell and the robot:

$$\underline{F}(i,j) = \frac{F_{cr} C(i,j)}{d^2(i,j)} \left[\frac{x_t - x_0}{d(i,j)} \dot{x} + \frac{y_t - y_0}{d(i,j)} \dot{y} \right] \quad (1)$$

where

F_{cr} = Force constant (repelling)
$d(i,j)$ = Distance between cell (i,j) and the robot
$C(i,j)$ = Certainty level of cell (i,j)
x_0, y_0 = Robot's present coordinates
x_i, y_j = Coordinates of cell (i,j)

Notice that in Eq. 1 the force constant is divided by d^2. By this method, occupied cells exert strong repulsive forces when they are in the immediate vicinity of the robot, and weak forces when they are further away.

The resultant repulsive force, $\underline{F}_r$, is the vectorial sum of the individual forces from all cells:

$$\underline{F}_r = \sum_{i,j} \underline{F}(i,j) \quad (2)$$

At any time during the motion, a constant-magnitude attracting force, $\underline{F}_t$, pulls the robot toward the target. $\underline{F}_t$ is generated by the target point T, whose coordinates are known to the robot. The target-attracting force $\underline{F}_t$ is given by

$$\underline{F}_t = F_{ct} \left[\frac{x_t - x_0}{d(t)} \dot{x} + \frac{y_t - y_0}{d(t)} \dot{y} \right] \quad (3)$$

where

F_{ct} = Force constant (attraction to the target)
$d(t)$ = Distance between the target and the robot
x_t, y_t = Target coordinates

Notice that $\underline{F}_t$ is independent of the absolute distance to the target.

The vectorial sum of all forces, repulsive from occupied cells and attractive from the target position, produces a resultant force vector $\underline{R}$:

$$\underline{R} = \underline{F}_t + \underline{F}_r \quad (4)$$

The direction of $\underline{R}$, $\delta = \underline{R}/|\underline{R}|$ (in degrees), is used as the reference for the robot's steering-rate command Ω

$$\Omega = K_s[\delta \ (-) \ \theta] \quad (5)$$

where

K_s = Proportional constant for steering (in sec^{-1})
θ = Current direction of travel (in degrees)

(-) is a specially defined operator for two operands, α and β (in degrees), and is used in the form $\delta = \alpha$ (-) β . The result, δ (in degrees), is the shortest rotational difference between α and β. Therefore, δ is always in the range $-180^o < \delta < 180^o$.

A typical obstacle map in Certainty Grid representation shows obstacle-boundaries as clusters of cells with high certainty values. Misreadings, on the other hand, occur at random and therefore produce mostly isolated cells with low certainty values. Summation of repulsive forces from occupied cells (Eq. 2) makes the robot highly responsive to clusters of filled cells, while almost completely ignoring isolated cells.

Our VFF algorithm has several advantages over edge-detection methods:

1. In-edge detection methods, misreadings or inaccurate range measurements may be misinterpreted as part of an obstacle, thereby gravely distorting the perceived shape of the obstacle. The sharply defined contour required by these methods cannot rectify the blurry and inaccurate information provided by the sensors. In our method, on the other hand, misreadings gradually deteriorate. The VFF concept does not utilize sharp contours in the world model (since most sensors do not produce adequate data for sharp contours), but responds to clusters of high likelihood for the existence of an obstacle.

2. Our force-field method does not require the robot to stop for data acquisition and evaluation, or for corner negotiation (as is the case in Crowley, 1984; Weisbin et al., 1986; Brooks, 1986; Borenstein and Koren, 1988). Ideally, our method would allow the robot to negotiate all obstacles while it travels at its maximum speed.

3. Updating the grid-map with sensor information and using the grid-map for navigation are two completely independent tasks that may be performed asynchronously, each at its optimal pace. The edge-detection method, in contrast, always requires the following activities to be performed in sequence: detect an obstacle, stop the robot, measure the obstacle (find its edges), recalculate the path, and resume the robot's motion.

4. The grid representation for mapping obstacles lends itself easily to the integration of data from other sensors (such as vision, touch, and proximity), in addition to data from previous runs or from preprogrammed obstacles (such as walls). Indeed, this navigation algorithm is altogether indifferent to the source of the map data, since different levels of confidence are expressed numerically.

Fig. 1 shows a run of the robot with actual ultrasonic data, obtained in real-time during the robot's motion. The robot ran at a maximum speed of 0.78 m/sec and achieved an average speed of 0.53 m/sec. The maximal range for the sensors was set to 2 m, which is why only part of the rightmost wall is shown, whereas the rear wall and most of the leftmost wall remained undetected. Each dot in Fig. 1 represents one cell with a Certainty Value (CV) unequal to zero. CVs are color-coded on the computer screen, but this can not be reproduced here. At least two misreadings can be identified in Fig. 1, which have been encircled for emphasis.

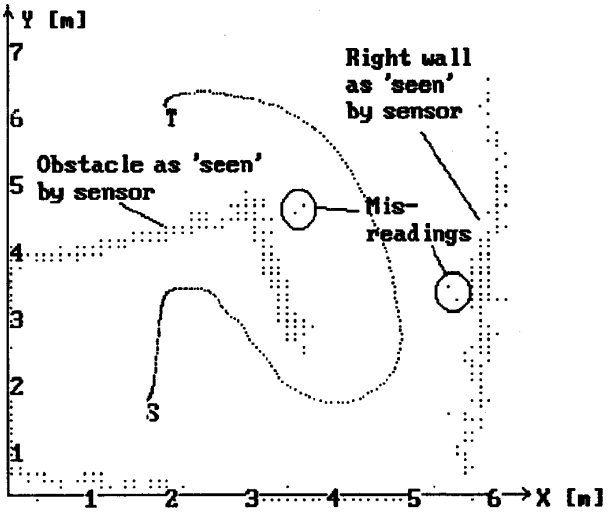

Fig. 1:
Robot run with actual ultrasonic data obtained in real-time during the robot's motion. Maximum speed = 0.78 m/sec and average speed = 0.53 m/sec.

4. Conclusions

A comprehensive obstacle avoidance approach for fast-running mobile robots, denoted as the VFF method, has been developed and tested on our experimental mobile robot CARMEL. The VFF method is based on the following principles:

1. A Certainty Grid for representation of (inaccurate) sensory data about obstacles provides a robust real-time world model.

2. A field of virtual attractive and repulsive forces determines the direction and speed of the mobile robot.

3. The combination of 1. and 2. results in the characteristic behavior of the robot: The robot responds to clusters of high-likelihood for the existence of an obstacle, while ignoring single (possibly erroneous) data points.

5. References

Borenstein, J., 1987, "The Nursing Robot System." Ph.D. Thesis, Technion, Haifa, Israel.

Borenstein, J. and Koren, Y., 1988, "Obstacle Avoidance With Ultrasonic Sensors." IEEE Journal of Robotics and Automation., Vol. RA-4, No. 2.

Brooks, R. A., 1986, "A Robust Layered Control System for a Mobile Robot." IEEE Journal of Robotics and Automation, Vol. RA-2, No. 1, pp. 14-23.

Brooks, R. A. and Connell, J. H., 1987, "Asynchronous Distributed Control System for a Mobile Robot", Proceedings of the SPIE, Vol. 727, Mobile Robots (1986), pp. 77-84.

Crowley, J. L., 1984, "Navigation for an Intelligent Mobile Robot." Carnegie-Mellon University, The Robotics Institute, Technical Report, August.

Elfes, A., 1985, "A Sonar-Based Mapping and Navigation System." Carnegie-Mellon University, The Robotics Institute, Technical Report, pp. 25-30.

Khatib, O., 1985, "Real-Time Obstacle Avoidance for Manipulators and Mobile Robots." 1985 IEEE International Conference on Robotics and Automation, March 25-28, St. Louis, pp. 500-505.

Krogh, B. H., 1984, "A Generalized Potential Field Approach to Obstacle Avoidance Control." International Robotics Research Conference, Bethlehem, PA, August.

Krogh, B. H. and Thorpe, C. E., 1986, "Integrated Path Planning and Dynamic Steering Control for Autonomous Vehicles." Proceedings of the 1986 IEEE International Conference on Robotics and Automation, San Francisco, California, April 7-10, pp. 1664-1669.

Moravec, H. P. and Elfes, A., 1985, "High Resolution Maps from Wide Angle Sonar." IEEE Conference on Robotics and Automation.

Moravec, H. P., 1986, "Certainty Grids for Mobile Robots." Preprint of Carnegie-Mellon University, The Robotics Institute, Technical Report.

Thorpe, C. F., 1985b, "Path Relaxation: Path Planning for a Mobile Robot." Carnegie-Mellon University, The Robotics Institute, Mobile Robots Laboratory, Autonomous Mobile Robots, Annual Report 1985, pp. 39-42.

Weisbin, C. R., de Saussure, G., and Kammer, D., 1986, "SELF-CONTROLLED. A Real-Time Expert System for an Autonomous Mobile Robot." Computers in Mechanical Engineering, September, pp. 12-19.

MAINTAINABILITY ANALYSIS FOR INTELLIGENT CONTROLS

Dev Raheja
Technology Management, Inc.
9811 Mallard Drive, Suite 213
Laurel, Maryland 20708 USA

Gita Raheja
Patni Computers, Bombay, India

Introduction

An intelligent software not only uses logical experience but also overcomes the weaknesses in the hardware/software interface. It knows how the hardware will fail and prepares to assume the most efficient control. This paper covers the software analysis methodology to minimize the downtime from software as well as the vulnerable hardware.

The downtime can result from the software's inherent design. If one looks at the costs of downtime from software, it is in multi proportion to downtime costs due to hardware. It is understandable that the invisible nature of software makes it difficult to troubleshoot, but the costs today are dangerously out of control. Some reasons are: incomplete specifications, absence of design for maintainability, and poorly structured programming. The most ignored element in this list is design for maintainability.

Software Maintainability - What is it ?

Software maintainability is the characteristic of software design and can be expressed as the probability that the downtime caused by software error shall be no more than the specified time when maintenance action is performed in accordance with the prescribed procedures and resources. Some try to include the ability to modify the software as part of maintainability but then they are really referring to "Adoptability". Maintainability is generally not quantified because the above definition contains several qualitative factors. It assumes the resources such as troubleshooting procedures are standardized and that the trained manpower is available all the time. Some parameters related to maintainability such as maximum downtime, and software recovery time, and inherent availability (percent uptime) can be specified.

Reducing Downtime Inherently in Design

The problem of measuring maintainability is larger. The software error once removed, is not expected to reappear except from a different cause. And if the software engineer knew about the error in advance, he/she will correct it at the outset. In short, the error removal time is unpredictable. The software engineer's answer to maintainability then is to use proactive analyses in predicting the possible type of errors and design the software to locate and fix them fast. Typically, the errors and their proportions could be the following as reported by Chenoweth and Schulmeyer (IEEE COMPSAC 86 Proceedings):

Logic errors	21.29 percent
Input/Output errors	14.74 percent
Data handling errors	14.49 percent
Computational errors	8.34 percent
Preset database errors	7.83 percent
User interface errors	7.70 percent
Documentation errors	6.25 percent
Routine to routine interface errors	5.62 percent

In this list the logic errors are 21.29 percent but actually they are much higher. These errors are hidden in the other catagories and may account for as much as 60 percent. Not included in this list are environmental failures from hardware causes and human errors. A maintainable software is therefore designed to repair fast inspite of logic and other forms of errors. Let's look at some design features required that should reduce downtime should an error occur.

Modular Design: will isolate the error to a module level and help reduce downtime from going through the entire software.

Structured Program: will offer clues to errors faster than unstructured program. The top-down structure may even speed up the process of fault isolation.

Testable Design: will be able to test important paths which may be not observable otherwise.

Fault Tolerant Design: will often incorporate triple modular redundancy. It will avoid downtime conceptually and the software will work as long as at least one module works. The designs could include similar features such as fault masking and fault avoidance.

0-8186-2012-9/89/0000/0385$01.00 © 1989 IEEE

Self Testing Design: can perform internal checks. IBM 3081 mainframe has a maintenance processor inside the host computer to monitor software and the hardware parameters.

Unambiguous System Requirements: will incorporate specifications based on a thorough analysis of predicted downtimes. It will include limits on downtime, and will incorporate most of the features mentioned above. System Requirements Definition is the first place and the most important one where maintainability effort is most productive.

Above design features form the core requirements to reduce downtime from logic errors. Other requirements may be added. A similar listing of requirements should be developed to quickly recover from coding errors. For example, high level languages and reusable software are used to avoid or prevent coding errors (best way to recover from an error is not to have one). Requirements such as signature analysis can be used to isolate certain coding errors. Similarly, error detection codes can be used to quickly identify input/output errors, and inclusion of the "retry" operation can overcome erroneus outputs due to voltage transients.

Using Analysis to Overcome System Vulnerability

This method makes use of systematic analysis such as Failure Modes, Effects and Criticality Analysis (FMECA) or equivalent methodology. The purpose is to predict missing requirements that could help in reducing the downtime. For example: Let us suppose that a software activates a red light bulb to indicate a dangerous condition. It is likely that the light bulb itself could burn out. If the software is not coded to detect the bulb failure, then there can be an enormous downtime or even a permanent damage to the system. This then becomes the missing requirement in the specification. The software FMECA, sometimes called FMEA, can overcome many similar deficiencies. The case history below will illustrate the use of this technique.

The Procedure

Hardware is inherently vulnerable to component failures. An intelligent software should be designed to anticipate such failures and take corrective actions. But the most difficult task is(for the software designer)to predict failures of the hardware. Fortunately, there are tools for predicting malfunctions. One such tool is Software Failure Mode, Effects, and Analysis (SFMEA). Below is the description of this technique.

The analysis is done as soon as the first draft of the System Requirements Specification is complete. No coding should be done unless this analysis is complete because many missing requirements are likely to be revealed during this analysis. This is a systematic analysis of the software flow logic. It identifies what could go wrong during each software task and what kind of design action should be taken to build the intelligence in the software to avoid downtime. The steps are (a) construct the program instruction flow chart or pseudo code from the System Requirements Definition. (b) document maintainability requirements, and (c) use SFMEA table to identify new requirements to overcome hardware vulnerability.

Constructing the Program Instruction Flow Chart

Figure 1 shows an overhead conveyor system for an automated material handling process. The system description is as follows: The plastic bucket comes in empty on the conveyor. Each bucket lands on a sensor which activates the drum to tilt. The bucket waits for 'n' seconds during which the food product is filled in the bucket. The bucket then goes to a scale which controls the product inventory. A bar coder then codes the bucket for its destination to a designated production line. Finally, the bucket passes through a dispatcher device which reads the bar code and routes the bucket. The entire process is controlled by the software. The objective is to prevent the process from stopping and prevent food from spilling over the buckets.

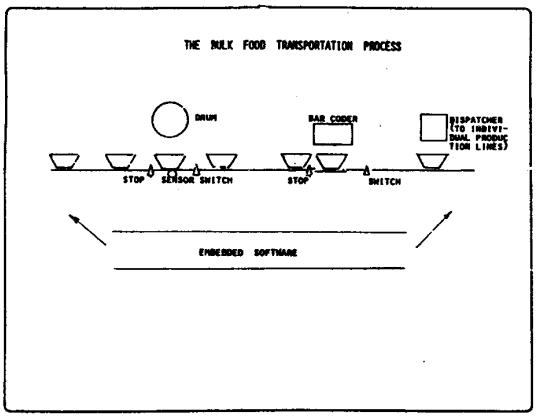

Figure 1. The Process Description

Documenting Maintainability Requirements

Unless the maintainability parameters are included in the System Requirements Specifications, they are likely to be ignored. In this case, some requirements are:

- .The software shall manage the process such that no product shall spill from too much product in the bucket or from a malfunction of the system (All spills require the process to shut down).
- .The joint availability of the software/hardware system shall be at least 95% (Downtime no more than 5%). The software shall exercise intelligent controls to prevent downtime from hardware malfunctions.

.The software shall perform self-tests to assure reliability each time the process starts after a shutdown.

.The software shall isolate all critical and major faults to avoid the time required to diagnose them. Such faults shall be identified in the Software Railure Mode and Effects Analysis, as catagory I and II items.

Software Failure Mode and Effects Analysis (SFMEA)

This is a systematic analysis to uncover weaknesses and missing requirements in the software specifications. Each task in the software is analyzed for what can go wrong while the task is being performed. The critical questions are:

.What can go wrong with the hardware?

.Is software capable of switching to an alternate mode or appropriate corrective action to prevent downtime?

If the answer to the later is 'No', the weakness in the software intelligence is obvious. In other words, some requirement in the software should have been there, but it is missing.

Figure 2 shows the flowchart. It shows all the software tasks sequentially. Each task is assigned a number for traceability.

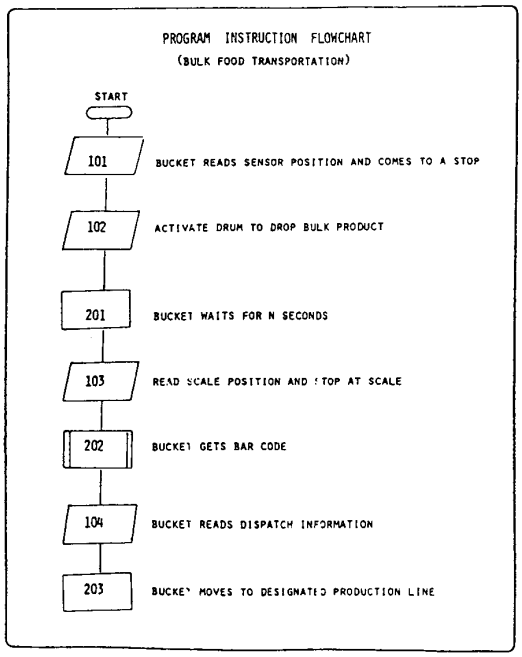

Figure 2. Program Instruction Flow

Figure 3 shows a portion of SFMEA with an explanation below.

Column 1 shows each software task.

Column 2 shows system vulnerability (failure mode) including the vulnerability of the software due to coding errors, input/output errors, and the program changes.

Column 3 contains the effects of vulnerability on the system.

Column 4 shows how critical the value of downtime is. In this table, catagory I stands for unacceptable downtime, II for major downtime, III for minor downtime, and IV for acceptable downtime.

Column 5 contains the recommendation after a careful review of the design. It identifies what new task the software will perform to avoid system downtime. A change in the hardware design may also be required. To prevent inherent downtime from software, one should question the possibility of input/output errors, data handling errors, and user interface errors. Software design may need changes to avoid downtime from these errors.

Examples of Developing New Requirements

Let's look at task 101. The bucket stands on the sensor and a software command is expected to halt the conveyor. The failure mode (vulnerability) is that the software gives command to stop, but the wrong bit winds up in the memory. As a result the conveyor will not stop and the buckets may pile up on each other further in the process. Column 4 shows that the criticality is not very high beacuse the buckets are empty but the downtime could be substantial. Column 5 shows that the software needs to check that the bucket is indeed full after it left the sensor and consider this feedback before more buckets are allowed to go. To minimize downtime, the software must also isolate this fault and issue a print statement to maintenance personnel. A more intelligent software will automatically switch over to a redundant sensor and assure quickest repair of the failed sensor. This will call for some additional hardware also.

Let's look at one more task: task 201. In this task the bucket is expected to wait 'n' seconds. The next column questions "What Can Go Wrong", while software is performing this task (failure mode). One failure mode is that the timer may malfunction and the software does not know it. The "effect" is that the product will overfill the bucket and spill. This becomes a potential health hazard because someone can put the contaminated product back into the bucket.

SOFTWARE FAILURE MODE AND EFFECTS ANALYSIS (SFMEA)

PROGRAM INSTRUCTION #		FAILURE MODE AND CAUSES	EFFECTS ON SYSTEM	CRITI-CALITY	DESIGN CHANGE
101	SOFTWARE READS BUCKET POSITION ON SENSOR AND STOPS BUCKET	WRONG BIT IN THE MEMORY POSITION	BUCKETS MOVE EMPTY AND PILE ON EACH OTHER	II	INCORPORATE ERROR DETECTION CODE AND GIVE A PRINT COMMAND
102	ACTIVATES DRUM TO DROP BULK PRODUCT	TILT MECHANISM SWITCH MAY NOT WORK DUE TO WRONG BIT IN MEMORY	BUCKET REMAINS EMPTY	IV	NO ACTION REQUIRED AT THIS STATION, IMPROVE TASK 201 TO PREVENT BUCKETS FROM PILING
		TILT MECHANISM SWITCH MAY FAIL AFTER THE DRUM IS TILTED	PRODUCT OVER-SPILLS ON PRODUCTION FLOOR	I	DESIGN ADDITIONAL SWITCH, SOFTWARE TO FLAG FAILURE OF ANY SWITCH
201	BUCKET WAITS FOR N SECONDS	BUCKET WAITS LONGER BECAUSE OF CLOCK MALFUNCTION	PRODUCT SPILLS	I	SOFTWARE TO CHECK CLOCK PERFORMANCE WITH A REDUNDANT CLOCK AND PRINT IF DEVIATION $\pm$ 2 SECONDS
		WRONG COMMAND TO CLOCK BECAUSE OF VOLTAGE TRANSIENT		I	INCORPORATE "RETRY"
103	READS SCALE EMPTY AND STOP BUCKET AT SCALE	SCALE MAY NOT BE EMPTY	PRODUCT SPILLS	I	PROVIDE FEEDBACK LOOP WHEN A BUCKET LEAVES THE SCALE

Figure 3. SFMEA Example

The criticality rank is I. The recommendation column now introduces a "new" requirement" that the software should constantly check the timer performance. Such requirements are sooner or later introduced out of necessity, but it is cheaper to put this missing requirement in early design. By the time this analysis was complete, more than 200 software requirements were added to the specification.

Conclusion

Software maintainability can use a lot of ideas from hardware maintainability. One should note that an analysis like this lets the software engineer ask structured questions on vulnerability but identifying additional requirements depends on the individual. It is, therefore, possible to still miss many design requirements. A critical software design should include some more proactive analyses such as fault tree analysis and sneak circuit analysis.