DOI 10.31256/UKRAS19.8

Enabling functional resilience in autonomous multi-arm and multi-vehicle cooperative tasks

Amar Kumar Behera, *Member, UKRAS*

***Abstract*— This paper presents results from experiments aimed at creating a framework for designing functionally resilient multi-robot systems. This is achieved by embedding functional information in motion planning algorithms and linking it to the robot's morphology. Two specific use case scenarios are discussed, one pertaining to multi-arm co-operative tasks and the other involving multi-vehicle tasks. Initial experimental results for each use case scenario are presented. The results indicate that the speed of response in the event of a disaster is dependent on the noise in the environment, processing power and intelligent mapping of functions to morphologies.**

I. Introduction

A team of robots or robotic manipulators performing tasks jointly either co-operatively or competitively have come to be defined as multi-robot systems. These systems can be thought of as complex engineered systems as they tend to display complex behavior, have a life of their own and can be hard to interpret analytically [1]. As such, there is often limited understanding regarding how such systems will function and behave in the real world and this can lead to operational difficulties in the form of cost overruns, delays in project completion and delivery, unplanned repair and maintenance and total system failure [2].

The design of multi-robot systems to meet performance specifications given by measures of predictability, reliability, stability, controllability and precision requires enabling resilience in these systems as they function under myriad environmental conditions and perform different tasks. Resilience can be defined as the ability of a system to autonomously recover when subjected to change, especially from disastrous events [3]. This, in turn, requires addressing the twin attributes of reliability and restoration [4]. While current robotic systems may exhibit resilience to a certain degree, systematic studies that discuss and enable functional resilience are lacking in literature and have been limited to very specific topics such as security [5] and coordination [6].

This work seeks to create an initial framework for functional resilience in multi-robot systems by taking an experimental approach where the requirements for resilience are derived from specific experimental scenarios involving change and disaster. These requirements are then used to embed intelligence within the software algorithms in the multi-robot system that enable the multi-robot system to function in a resilient manner.

II. Methodology

Functional resilience was explored in the context of two types of multi-robot systems: i) multi-arm and ii) multi-vehicle. These are discussed below.

A. Multi-arm experiment

A multi-arm experiment was set up on a Baxter collaborative robot. The task given to the robot was to pick and place pegs and rings on a Bytronic Industrial Control Trainer (ICT3) conveyor belt. The trainer is designed to assemble the pegs and rings. The pegs are made of aluminium alloy while the rings are made of white-colored polymeric material. The pegs and rings are placed in separate red and green bins respectively, as shown in Fig. 22. The task given to the left arm was to place the pegs on the conveyor, while the right arm was required to place the rings on the conveyor. Pick and place operation on the Baxter can be performed using visual servoing using cameras on the robot. A specific demo example on golf balls was calibrated for use in this study.

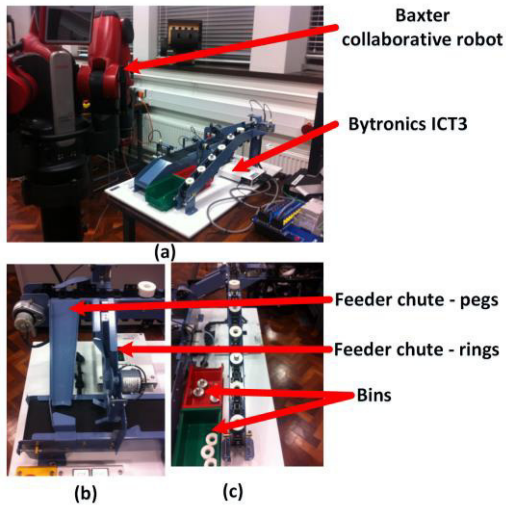

Fig. 22. Multi-arm experiment showing (a) entire setup (b) feeder chutes of Bytronic ICT3 (c) bins with conveyor belt

The following scenarios for functional resilience were conceived:

- If the left arm fails, then the right arm takes over the entire pick and place operation for both pegs and rings
- If the right arm fails, then the left arm takes over the entire pick and place operation for both pegs and rings

*Research is supported by Engineering Complexity Resilience (ENCORE) Network+ grant.

A.K.Behera is with the Centre for Intelligent Autonomous Manufacturing Systems (i-AMS), Queen's University Belfast, Belfast BT9 5AH (phone: +44 28 9097 4769 e-mail: a.behera@qub.ac.uk).

- If both arms fail, then an error message is displayed stating that both arms are dysfunctional
- If any one or both of the arms are functional again, the robot returns back to working with one or both arms

An algorithm was written to realize the above scenario that enables functional resilience by embedding intelligence into the code for the pick and place of pegs and rings.

B. Multi-vehicle experiment

A multi-vehicle experiment was set up using a commercially available unmanned ground vehicle (UGV) platform, Diddyborg together with an unmanned aerial vehicle (UAV) platform, CoDrone Pro, as shown in Fig. 23. A search and inspect experiment was designed where the UAV-UGV combination is sent out to autonomously search an area for a red ball and once the ball is found, the drone takes off to inspect the area. The Diddyborg is equipped with a Raspberry Pi 3 model B+ board and camera. The autonomous following of the ball is achieved using a Python code, adapted from a demo example, that uses the OpenCV library to process the images from the Raspberry Pi camera and detect objects of a specific color and shape. A disaster scenario is introduced in the shape of a blue obstacle that appears suddenly in the field of view of the UGV. Resilience is to be achieved so that the obstacle can be identified in real time, the multi-robot system stops a certain distance before the obstacle and the drone takes off to inspect the area following which a new path is charted for achieving the task of finding the red ball.

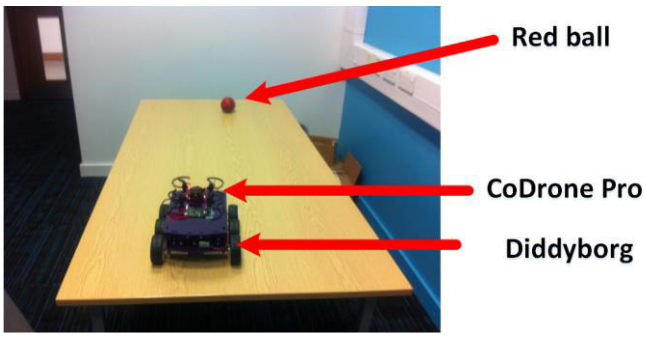

Fig. 23. Multi-vehicle experiment with an UAV and UGV performing a search and inspect operation

III. RESULTS

This section discusses i) how the scenarios for functional resilience were embedded in the code of the robots and ii) the robustness of the algorithms embedding the resilience.

A. Enabling functional resilience in multi-arm experiment

A careful survey of programming techniques that would be suitable for enabling resilience led to the conclusion the key to enabling functional resilience is to be able to run multiple processes in parallel. For instance, while a robot is carrying out a certain task using a function or a set of functions called from the main function, if a simultaneous check could be run using another function whether all the manipulating arms are functional by processing sensor data, then if it found that one of the arms is failing, the other manipulating arms could take over the task or a remedial measure be put in place for the faulty arm. Having identified multi-threading as the technique to be used to enable resilience, the next step was to artificially inject one of the four scenarios identified in Section II.A. This was done by using a mouse click. A click of the left mouse button was meant to indicate that the left arm was non-functional while a click of the right mouse button made the right arm non-functional. The pseudo code that enables resilient operation is listed below –

Algorithm: Multi-arm resilience using multiple threads

Input: Adverse and repair events as mouse button clicks

Output: Pick and place using functional arm(s) or display error message

```
1. Obtain current state of mouse buttons as 'state_left' and 'state_right'
2. Define global variables 'count_left_mouse', 'count_right_mouse' that count the
number of clicks made with each mouse button so far
3. Define start_robot as a flag for thread that kickstarts robot with both arms functioning
4.
5. # Define a function for pick and place with both arms
6. def pick_place_both():
7.     count = 0 # variable to keep track of maximum number of rings that can be fed to feeder chute
8.     while count < max_feed and count_left_mouse % 2 == 0 and count_right_mouse % 2 == 0:
9.         count + = 1
10.        pick_bin_B1(left_arm)
11.        pick_bin_B2(right_arm)
12.        time.sleep(10) # time required to perform pick and place operations
13.

14. # Define a function for pick and place with left arm
15. def pick_place_left():
16.    count = 0
17.    while count < max_feed and count_left_mouse % 2 == 0 and count_right_mouse % 2 != 0:
18.        count += 1
19.        pick_bin_B1(left_arm)
20.        pick_bin_B2(left_arm)
21.        time.sleep(10)
22.
23. # Define a function for pick and place with right arm
24. def pick_place_right():
25.    count = 0
26.    while count < max_feed and count_left_mouse % 2 != 0 and count_right_mouse % 2 == 0:
27.        count += 1
28.        pick_bin_B1(right_arm)
29.        pick_bin_B2(right_arm)
30.        time.sleep(10)
31.
32. # Define a function for both arms dysfunctional
33. def pick_place_error():
34.    count = 0
35.    while count < max_feed and count_left_mouse % 2 != 0 and count_right_mouse % 2 != 0:
36.        count += 1
37.        print("Both arms dysfunctional \n")
38.        time.sleep(10)
39.
40. while True: # loop that enables continuous pick and place
41.    Obtain current state of left and right mouse buttons as 'a' and 'b'
42.
43.    if a != state_left:  # Button state changed
44.        state_left = a
45.        count_left_mouse += 1
46.            if count_left_mouse % 2 != 0 and count_right_mouse % 2 == 0:
47.                    print("Left arm not functioning")
48.                    start_new_thread(pick_place_right,())
49.            elif count_left_mouse % 2 == 0 and count_right_mouse % 2 != 0:
50.                    print("Right arm not functioning")
51.                    start_new_thread(pick_place_left,())
52.            elif count_left_mouse % 2 == 0 and count_right_mouse % 2 == 0:
53.                    start_new_thread(pick_place_both,())
54.            elif count_left_mouse % 2 != 0 and count_right_mouse % 2 != 0:
55.                    start_new_thread(pick_place_error,())
56.

57.    if b != state_right:  # Button state changed
58.        state_right = b
59.        count_right_mouse += 1
60.            if count_right_mouse % 2 != 0 and count_left_mouse % 2 == 0:
61.                    print("Right arm not functioning")
62.                    start_new_thread(pick_place_left,())
63.            elif count_right_mouse % 2 == 0 and count_left_mouse % 2 != 0:
64.                    print ("Left arm not functioning")
65.                    start_new_thread(pick_place_right,())
66.            elif count_right_mouse % 2 == 0:
67.                    start_new_thread(pick_place_both,())
68.            elif count_left_mouse % 2 != 0 and count_right_mouse % 2 != 0:
69.                    start_new_thread(pick_place_error,())
70.
71.    if start_robot == 0:
72.        start_new_thread(pick_place_both,()) # Runs only when kickstarting robot with both arms
73.        start_robot +=1 # Stops this thread from running after adverse events kick in
```

A series of timed tests were performed with a stop watch to assess the robustness of the multi-threading algorithm. Results from one such test are presented below in Table I.

TABLE I. STATISTICS FROM DISASTER EVENTS IN MULTI-ARM PICK AND PLACE USING A MULTI-THREADING ALGORITHM

Total time for experiment	5min 28 s
Time between events	16.4 s
Number of disaster events	21
Number of events with 0 subsequent error	2
Number of events with 1 subsequent error	17
Number of events with 2 subsequent errors	2

It may be noted that every pick and place event has a delay time of 10 s as mentioned in the pseudo code. This delay time is the time it takes to perform the operation. When a disaster event occurs, a pick and place event is already occurring and the thread performing it is running. Hence, even after the disaster event occurs, the operation is shown to being done and hence, an error occurs. This explains the 17 errors for the 21 disaster events. However, if two disaster events occur in close proximity of one another and negate the effect of each other (e.g. power supply fluctuations), then no error may be recorded. This explains the 2 cases with no subsequent errors. Another possibility is that the same thread is started twice due to such fast fluctuations leading to 2 subsequent errors.

B. Enabling functional resilience in multi-vehicle experiment

The first step was to pair the drone and the raspberry pi and fly it. Although the bluetooth chip on the pi (Bluetooth 4.2 chip Cypress CYW43455) communicates with the bluetooth chip on the drone (bluetooth 4.0 BLE), it was found that only pairing is feasible and flying the drone could not be enabled. Hence, the bluetooth board from CoDrone Pro had to be carried onboard and plugged in to the USB port of the pi. Next, the CoDrone library was imported within the diddyborg Python code, creating an instance of the drone, pairing it, and then having the drone take off after the diddyborg arrived at the red ball. The diddyborg has a thread where it prints a message "Close enough" on reaching the ball and this is where the drone take off event was created. A youtube video of the successful experiment is available at https://youtu.be/-Lo_2z0fhMg. The code was then modified to be able to detect blue objects in addition to red object, treat the blue objects as obstacle and inspect them. The pseudo code for the same is given below –

Algorithm: Multi-vehicle resilience using color-space based rules

Input: Streaming video of area being navigated
Output: Search and inspect red balls and blue obstacles

```
1. Instantiate an object of type CoDrone as "drone"
2. # Pair drone and bluetooth module connected to Raspberry Pi by
specifying USB port
3. drone.pair (drone.Nearest, USB port connected to Bluetooth module)
4. Blur image obtained from raspberry pi using the function 'medianBlur'
5. Change the image definition from RGB to HSV colorspace using the
function 'cvtColor'
6. Find the portion of the image in the red and blue channels using the
function 'inRange' between numpy arrays
7. Use the function 'findContours' to segment the image into red contours
and blue contours
8. Find the center of each set of contours and the area
9. Determine distance to blue obstacle and red ball using the centers and area
10. if (distance of obstacle to robot < distance of ball to robot)
11.     if (obstacle is in the path to the red ball)
12.         navigate close to obstacle
13.         fly drone to inspect obstacle
14.     else if (obstacle in not in the path to the red ball)
15.         navigate to red ball
16.         fly drone to inspect red ball
17. else if (distance of obstacle to robot > distance of ball to robot)
18.         navigate to red ball
19.         fly drone to inspect red ball
```

The algorithm used to estimate the distance of an object from the robot is based on an area calculation. The speeds of the motors on the robot are adjusted according to the location of the centers in two axes in the plane of the camera of the robot and the area calculation. While the threshold values for the red ball worked for the demo diddyborg code available from the manufacturer, the use of the drone necessitated an upgrade from Python 2.7 to Python 3 as the drone's libraries are written in Python 3. This meant the older Open CV libraries no longer worked and needed a fresh install. When this was done, the area thresholds for objects needed changing. Fig. 24 shows the new calibration that was done for blue objects. While earlier, the ball following code for the diddyborg worked well with a value of 10, 000 for autoMaxArea corresponding to a ball 65 mm in diameter, the new value for the same was re-adjusted to 1, 000 based on the below calibration. This produced satisfactory results for identifying both obstacles and balls.

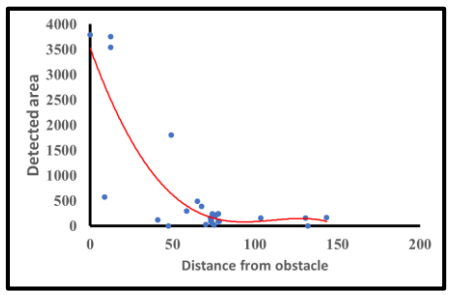

Fig. 24. Area of a blue object as a function of distance from camera; the red line shows a polynomial fit through the measured data points

IV. DISCUSSION

A discussion on the interpretation of the results from the multi-arm and multi-vehicle experiments is presented below.

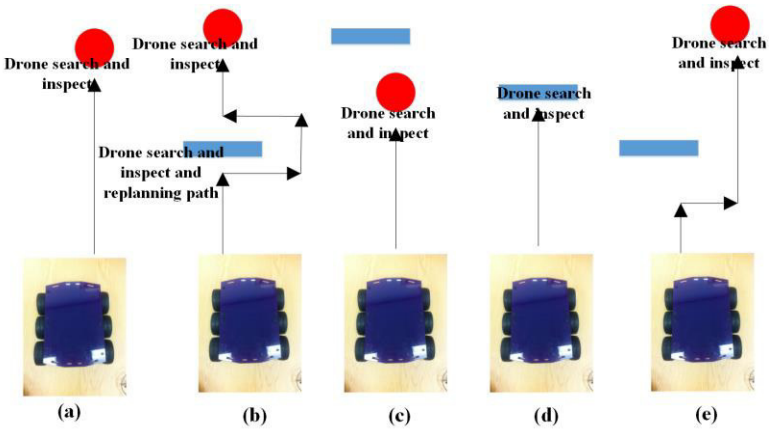

Fig. 25. Scenarios for obstacle emergence and associated path planning strategies a) no obstacle b) obstacle in line of sight of ball c) ball in front of obstacle d) only obstacle and no ball e) both ball and obstacle visible to the multi-vehicle team

A. *Discussion on multi-arm resilience experiment*

The introduction of a disaster scenario during the functioning of the multi-arm bin-picking experiment revealed that threads need to be handled with efficiency for the system to be resilient. Else, two situations may occur: i) delay in task allocation to the functional arm, ii) repetition of activities by a specific arm due to multiple instances of the same function running.

The first situation is hard to remedy as the non-functional arm may come in the way of the functional arm and an active collision avoidance algorithm between the two arms is necessary. Further, the thread for the non-functional arm needs to be interrupted immediately. Human intervention may be essential for the functional arm to take over the tasks or an intelligent algorithm to re-position the non-functional arm needs to be in place, which may be hard to realize, especially if the disaster scenario involves a power supply issue to the motors operating the non-functional arm.

The second situation can be remedied by introducing a waiting time between activities where the thread sleeps. Hence, a new pick and place activity does not start until the disaster scenario has kicked in. However, this will reduce the productivity from the multi-arm tasks and hence, was not pursued within this work.

A few hardware related issues need to be kept in mind before a fully resilient operation can be illustrated. One of these is being able to perform visual servoing on different types of objects with accuracy. These include the effects of color, size and shape, especially in different ambient conditions. Secondly, the placement of pegs and rings in the slots on the conveyor belt is not an easy task, and the positioning accuracy of the Baxter needs to be improved for the task to be performed precisely every time.

B. *Discussion on multi-vehicle resilience experiment*

The multi-vehicle experiment required the robot to identify new obstacles in its path as they emerge and inspect them. While an active collision avoidance algorithm can enable this, doing this based on color is significantly more challenging. Additionally, several scenarios for path planning can be envisaged as shown in the Fig. 4 below.

For the drone and the diddyborg to continuously operate as a team autonomously, the diddyborg must carry the drone with it at all times. This requires the drone to land on top of the diddyborg, after performing its search and inspect operation. This is easy to do using a joystick, however, achieving this autonomously requires the drone to run additional algorithms for detecting the diddyborg and making a stable landing on its top plate.

V. Conclusions

Functional resilience was explored experimentally in the context of multi-arm and multi-vehicle scenarios. The use of multi-threading in enabling resilience was outlined and algorithms for achieving this were successfully tested. Key challenges in achieving resilience in autonomous co-operative tasks were discussed, which provide the motivation for further research. Improvements in hardware and improved efficiency in the underlying algorithms can help create resilient multi-robot systems of the future.

Acknowledgments

The author wishes to acknowledge the contributions of Conor Burrows and Sarka Klimkova in building the robots and setup of experiments.

References

[1] A.A. Mina, D. Braha, Y. Bar-Yam, Complex engineered systems: A new paradigm, Springer, 2006.
[2] C. Ivory, N. Alderman, Can project management learn anything from studies of failure in complex systems?, Project Management Journal, 36 (2005) 5-16.
[3] R.J.T. Klein, R.J. Nicholls, F. Thomalla, Resilience to natural hazards: How useful is this concept?, Global Environmental Change Part B: Environmental Hazards, 5 (2003) 35-45.
[4] B.D. Youn, C. Hu, P. Wang, Resilience-driven system design of complex engineered systems, Journal of Mechanical Design, 133 (2011) 101011.
[5] S. Gil, S. Kumar, M. Mazumder, D. Katabi, D. Rus, Guaranteeing spoof-resilient multi-robot networks, Autonomous Robots, 41 (2017) 1383-1400.
[6] M.B. Dias, M. Zinck, R. Zlot, A. Stentz, Robust multirobot coordination in dynamic environments, in: IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA'04. 2004, IEEE, 2004, pp. 3435-3442.