Evolutionary Based Fault Injection for Multi-Robot Systems

Lenos Tsokkis
MSc Advanced Computer Science
University of York
York, United Kingdom

***Abstract*—Multi-robot systems capabilities are constantly evolving to meet the extensive need for intelligent and robust systems, which can accomplish high-cost and complex missions. Such systems are described by complex architectures which involve the cooperation of a vast number of components such as sensors, processing units, motion sources, actuators etc., thus making them highly prone to faults and errors, which can dramatically contribute to a disastrous outcome for a target mission. The key to avoid such catastrophic scenarios relies on both the extensive engineering of the multi-robot system's architecture as well as the definition and analysis of a pool, populated by an extensive number of possible fault/error variations related to the system's components operation and intercommunication. The current challenge it is aimed to be addressed by introducing a system defined by the model-driven engineering approach, which utilizes domain-specific languages for the creation of the mission and fault specification pool, model-to-text transformation for code generation related to the architecture of the robotic team, as well as evolutionary computation techniques for the evolving of fault specification variations, which are then injected into the system through a fault injection engine. The system's capability of producing hazardous fault specifications for future analysis, is verified through the correlation between the generated by the genetic algorithm fault specifications, and the violations on a predefined set of mission goals.**

Keywords—multi-robot systems, genetic algorithms, model-driven engineering, fault injection, evolutionary computation, domain-specific languages

I. Introduction

Multi-robot systems are becoming an area of increased research interest. The ability of robotic teams to cooperate and execute missions often impossible for single robots, creates a broad domain of applications including underwater or difficult terrain missions as well as hazardous for the human missions such as inspection of areas with high radiation, data sample gathering from the high depths in the ocean etc. These missions always include actions such as detections, navigation, coordination, collective decision making, etc. Multi-robot systems excel on scenarios where fault tolerance and autonomy are a high priority as these teams cooperate in a way which enables task reassignment from a failed robot to a healthy member of the team as well as efficiently preserving resources, through efficient division of the workspace, according to several parameters such as environmental events, remaining energy, current position, and health of the robots.

Often MRS are described as complex systems with multiple components cooperating both internally and externally during a mission. This creates a broad domain of possible faults and errors regarding operation and communications, which can extensively contribute into unpredictable behaviours and possibly unwanted mission outcomes. Failure of identifying the weak spots and flaws within the system, as well as how the system will respond in case of a failure could lead to an extremely hazardous situation. This hazardous situation can massively contribute to mission failures, resources wasting, environmental damage and risk of catastrophic accidents. For that reason, there is an extensive need for implementing a strategy around identification and extensive analysis of these possible faults and unwanted scenarios, in order to mitigate and address the possible roots of causes, as well as focus on specific fault types and areas of interest, regarding mission goal violations (e.g., energy sufficiency and coordination goals). Introduction of fault injection mechanisms can greatly benefit identification, analysis, and prevention of such fault scenarios. This can be achieved through generating and injecting random fault scenarios on simulated multi-robot missions, which enables observance and evaluation of critical parameters such system's reaction on certain fault classes, classification of impact caused by different fault specifications, as well as improvements to the system's remediation mechanisms, with the ability to focus on specific areas of interest regarding mission goal violations, requirements, and resource management.

This paper introduces a designed system defined by the Model-driven engineering principles [1], which enables research on identification of critical fault specification variations through evolutionary based fault injection. The system comprises the Genetic Algorithm [2] approach, with the ability to focus on violation of specific type of mission goals and requirements. The system comprises two Domain Specific Languages (DSLs) to describe both mission specification including robot characteristics and components, mission goals and mission map as well as the pool of possible fault classes. Furthermore, the system uses a code generation engine which consumes a model generated using the two DSLs and produces the backbone infrastructure of the system including a middleware which implements the components defined within the model using the necessary pre-implemented Python classes as well as loaders generated through model-to-text transformations [3], a collective intelligence (CI) template, a simulation interface, and a log-based simulation. All these components synthesize a robust solution on evolutionary based fault injection for multi-robot systems. The remainder of the paper is structured as follows. Section II describes related work on the field of fault injection and analysis on multi-robot systems. Section III illustrates and analyses the system's architecture, with a reference on the general workflow as well as an analysis of the fundamental components of the system, including the DSLs, fault injection engine, middleware, genetic algorithm, simulation interface, collective intelligence (CI) algorithm and the ROS log-based simulation. Section IV gives a brief explanation of the developed prototype which is used on Section V for experiment and evaluation. Finally, Section VI summarises the results and proposes possible future work areas.

II. Related Work

The work presented in this paper, is inspired by the fault injection and analysis domain, in multi-robot systems. Identification, classification, and analysis of fault tolerance as well as remediation and recovery from faulty states within the system has been a popular domain of research including various studies and research on fault injection techniques and strategies, for robotic systems.

Favier et al [4], worked on implementing a fault tolerance strategy which utilises fault trees to identify fault tolerance mechanisms, and fault injection methods to verify them, with a case study which used the Gazebo [5] simulator.

Fault injection is a mandatory component on developing fault detection and treatment solutions. Garrido et al [6], as part of their study on increasing the reliability and safety of autonomous vehicles, they created a fault injection tool for Flight Simulator X [7], alongside with a classification system which is used to assist on development of fault detection and treatment system for aircraft controllers.

A model-driven approach combined with fault injection on simulated environment (Gazebo) and virtual robot is proposed by Uriagereka et al [8]. Specifically, the study focuses on fault injection through the development of elTUS (Experimental infrastructure Towards Ubiquitously Safe Robotic Systems) fault injection framework, which aims to assess safety of robotic systems in the early design phase, by identifying potential hazardous scenarios and evaluation of controller in respect of certain faults applied.

A study from Wotawa [9], focuses on fault injection in combination with combinatorial testing for both testing self-adaptive systems and reducing the number of test cases taken in account.

Machine learning and neural networks can be proven beneficial in the domain of fault injection for autonomous robot systems. A study from Jha et al [10], presents a machine learning-based fault injection engine named DriveFI, which can identify faults with maximum impact on autonomous vehicles, with an application on two AV technology stacks from NVIDIA and Baidu. DriveFI incorporates both Batesian and traditional fault injection frameworks. Christensen et al [11], focus on fault detection by proposing the usage of back-propagation neural networks [12] and fault injection, to develop fault-detection components for autonomous mobile robots. This method involves learning from examples where faults are injected as well as from situations where the robots operate normally. These fault detection components can be trained to detect various faults as well as enabling robots to detect faults occurring in other robots.

Fu et al [13], introduce a Fault Injection Framework (FIF) for hardware and software faults at runtime, validate on top of Hardware-In-the-Loop setups for autonomous driving utilising the NXP BlueBox prototyping platform. Furthermore, they developed an interactive user interface based on the Robot Operating System for triggering automated fault injection processes, as well as for providing vehicle health visualisation. The developed fault injection mechanism was able to identify safety flaws and properties on automotive systems.

Juez et al [14], propose a simulation-based fault injection approach which is used for identifying safety properties which can be used for model-based design of automotive systems. Specifically, they developed a prototype called Sabotage, which coordinates with the Dynacar vehicle simulator [15] in order to enhance the simulation with fault models which will further contribute to safety. In addition, the prototype can be

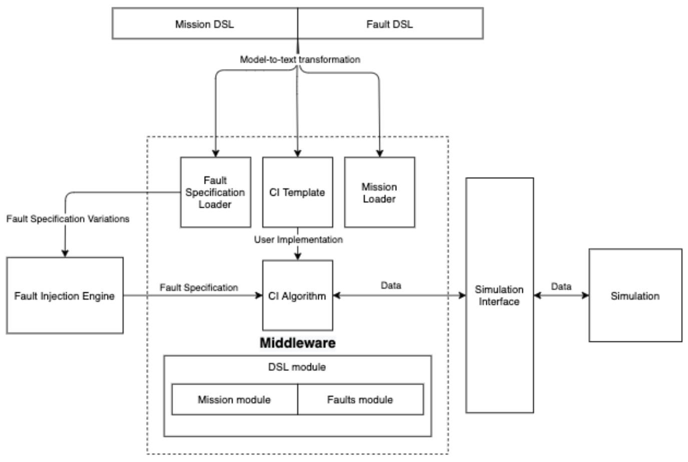


Fig. 1: System Architecture

used for actions such as analysing, executing, and configuring simulation's results. The approach was evaluated in a case study about the model-based design of a Lateral Control System and illustrates a major strength of evaluating system's safety during the early design phases.

III. System Architecture

A. General Overview

A general overview of the system's architecture and design is illustrated in Figure 1. The backbone of the system consists of several key components which necessary for the fundamental operation of the system. The components which compose the system's core are the 2 DSL's which are used for defining mission and robot team specifications as well as the pool of fault variations, a code generation engine which utilises the model-driven engineering principles [16], the middleware which provides the implementation (mission and fault specification related classes, CI, loaders etc.) of the logic for the biggest part of system's operation, a fault injection engine which implements a genetic algorithm and injects variations of fault specifications to the running simulation for evaluation, a simulation ROS [17] specific interface for enabling communication between the middleware and the running simulation, and a custom-made logging-based simulation which utilises the roslibpy [18] module for operation. The general workflow of the system can be described in the following steps:

*Step 1***.** Definition of mission and fault specification models through the use of DSLs.

Step 2. Consuming the model and the mission and fault Python modules, to generate the middleware code with all the necessary components including CI algorithm template, simulation interface, mission, and fault specification loaders as well as the ROS log-based simulation, through model-to-text transformations.

Step 3. Users implement the logic of the CI algorithm by filling the generated templates.

Step 4. Implementation of the genetic algorithm within the Fault Injection Engine and execution of evolution process.

Step 5. Analysis of the results and metrics produced by the CI algorithm, GA, and simulation log files.

Step 6. Identification of the most critical fault specification variations produced by the genetic algorithm.

B. DSLs

The system utilises 2 main custom-made DSL, Mission DSL, and Fault Specification DSL, which are developed using the Epsilon language family and specifically the Eclipse Modelling Framework (EMF) [19].

a) Mission DSL

The Mission DSL describes the specification of the mission, including all the definitions for all the fundamental components such as robots, servers, and their subcomponents (sensors, batteries, motions sources etc.), mission goals, obstacles, as well as other mission information such as mission duration and area. Some of the core classes of the mission DSL are further explained.

The *Robot* class as can be seen in Figure 2, is used to describe a robot including properties such as name, ID, speed, starting position, subcomponents, and user-defined properties of type String, Int or Double. Through the user-defined properties, a user can introduce custom made properties based on the robot specification needs. This introduces an extra layer of flexibility to the design and composure of the robot specifications. The selection of robot subcomponents is limited to the basic parts of a robot's specification such as batteries, sensors, and motion sources. Each of the subcomponent classes include basic information such as *ID*, name and *parentID* as well as more class-specific properties e.g., *sensorType*, *energyPerSample* and *samplesPerSecond* for the *Sensor* class, *totalEnergy* for the *Battery* class and *energyPerDistanceUnit* for the *Motion Source* class.

The *Goal* class represents the goals of the current mission and consists of properties such as *name*, *ID*, *members* property which defines the participants to that specific goal, *dependentGoals* to define other dependant goals, *area* which is the area in which the goal takes place, the related to a specific goal message, which are exchanged between robots and servers and the task associated with the specific goal. The *GoalTask* class instantiates the following goal tasks including, Patrol, *Visit*, *Track*, *Stay*, *AvoidCollision*, *GatherSamples*, *SufficientEnergy*, *StayWithinMissionArea and FixedDistanceBetweenRobots* tasks, which represent some of the core goals for a typical mission which can be also found implemented on various robot software development frameworks and operating systems, such as MOOS-IvP [20] and ROS [17]. The metamodel of the mission DSL can be easily modified to support more variations of goal tasks according to a mission's needs. The implementation of the goal tasks is left to the user as well as extensibility capabilities, by easily adding new goal-specific tasks.

The *Obstacle* class describes the obstacles placed in the mission area and consists of an *Area* property for the definition of coordinates and radius of the obstacle.

b) Fault Specification DSL

The Fault specification DSL is used to describe the pool of possible fault variations which compose a possible fault specification to be applied on a mission. This DSL consists of all definitions and properties of each individual fault type.

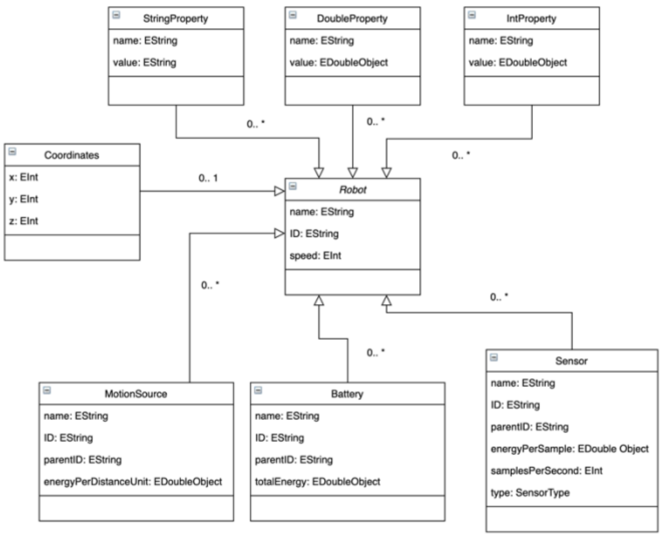
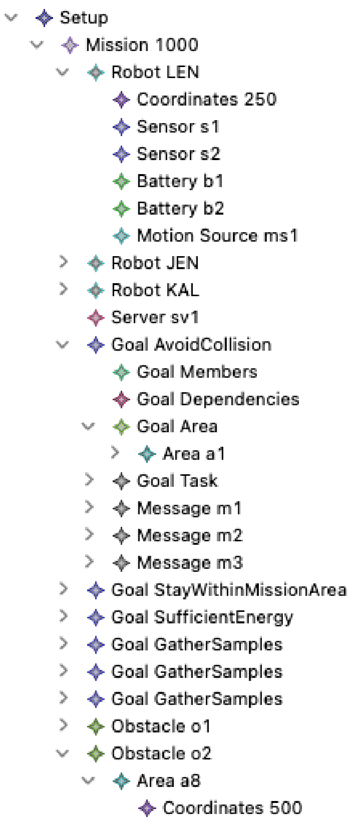


Fig. 2: Robot class

The *Fault* class includes all the fault types as properties as well as the name attribute to distinguish each individual created fault. Generally, fault types are separated in 2 main categories, the faults which affect messages exchanged between robots and servers during the simulation, and faults which will generate messages with unwanted commands and actions for the robots to execute. Faults on exchanged messages consists of many different types such as modification of exchanged data values (speed, energy, position etc.), as well as modification of the current local mission state on the offshore server such as obstacles positions etc. Modifying messages includes altering messages related to a robot's current speed, current



- Setup
  - Mission 1000
    - Robot LEN
      - Coordinates 250
      - Sensor s1
      - Sensor s2
      - Battery b1
      - Battery b2
      - Motion Source ms1
    - Robot JEN
    - Robot KAL
    - Server sv1
    - Goal AvoidCollision
      - Goal Members
      - Goal Dependencies
      - Goal Area
        - Area a1
      - Goal Task
      - Message m1
      - Message m2
      - Message m3
    - Goal StayWithinMissionArea
    - Goal SufficientEnergy
    - Goal GatherSamples
    - Goal GatherSamples
    - Goal GatherSamples
    - Obstacle o1
    - Obstacle o2
      - Area a8
        - Coordinates 500

Fig. 3: Mission model used in Section V

energy, current position, and parameters related to sensors such as *energyPerSample*, *sampleRate* etc. Some of the altering actions on messages include incrementing, decrementing, and setting random or fixed values to the robot properties carried in the messages which are exchanged between the systems components. Each fault type class consist of a reference to the affected message defined within the Mission DSL, as well as attributes related to the new value which is going to be assigned to the exchanged message.

A representation of the model generated by the 2 DSLs and used in section V can be seen in Figure 3. The model defines a mission with three robots named (LEN, JEN and KAL) and their subcomponents, a server for the CI execution, 6 mission goals, 8 obstacles placed within the mission area map and a fault specification which consists of 30 fault variations.

C. Middleware

The middleware is the core component of the designed system which holds the backbone of the whole infrastructure's implementation. The middleware includes all the necessary Python classes which implement the components of both mission and fault specification DSLs. These classes are split into 2 hand crafted modules, the mission and the faults module which are used explicitly for the system's operation, based on the specifications describe by the DSLs, and are used for the software implementation of the mission, fault specification, simulation etc. Additionally, the middleware incorporates the CI algorithm and the Fault Injection Engine which are analysed on the next sections. Furthermore, the middleware includes 2 loaders, the mission loader and the fault specification loader that are generated using model-to-text transformations which combine Python and EGL [3] code, and utilise the model shown in Figure 3.

a) Mission loader

The mission loader is responsible to implement every feature of the mission specification, including robot objects, goals, messages, obstacles etc. by encapsulating them into a single mission instance generated using the classes within the mission DSL module. This loader will be later used by the CI algorithm to generate the mission object which will be used as a local representation of the mission's state.

b) Fault specification loader

The fault specification loader is responsible to instantiate all the fault objects using the Python classes within the fault module. The loader comprises the fault specification model created by the user, creates the fault objects, and assigns to them random start and finish times, ensuring that faults which affect the same messages do not overlap in order to avoid invalid states where for example both *ActivateSensor* and *DeactivateSensor* faults are applied to the same sensor component.

Middleware is an important part of the whole system's architecture due to enabling separation of concerns between the endpoints of CI, Fault Injection Engine, and the running simulation. The CI and simulation exchange information via a simulation interface inside the middleware, thus there are no dependencies between the selection of simulation or CI algorithm. The middleware exposes through the simulation interface, various connection endpoints for both publishing and subscribing data, by utilising the roslibpy and the publish-subscribe architecture [21]. This creates two independent sides, enabling the user to experiment on different CI variations, without needing to modify or redesign the way CI algorithm communicates with the simulation regarding decision making.

D. Fault Injection Engine

The Fault Injection Engine is an independent component of the whole system which utilises the *Fault Specification* class to generate fault specification variations which are going to be used as an input population to the genetic algorithm. Fault Injection Engine is the connecting component between the collective intelligence algorithm (CI) and the genetic algorithm (GA). It encapsulates the implementation of the selected GA which is implemented using the DEAP library [22], with all the necessary functions such as mutation, crossover, fitness evaluation etc. The genetic algorithm and the related functions are all implemented by the user to allow more flexibility to the design without affecting any other component of the designed system. This enables users to easily deploy and experiment on different variations of genetic algorithms and their parameters, in order to find the optimal GA configuration for a specific use case. The main key operations of the Fault Injection Engine are the generation of the fault specification variations, the execution of the evolution process through the execution of the genetic algorithm in cooperation with the CI algorithm and the ROS simulation, as well as the generation of metrics and plots related to various evolution parameters which can be used for experimental analysis purposes (see section V) and can be extracted easily using features of the DEAP library. A full explanation of the selected GA architecture is explained in the next section.

E. Genetic Algorithm

The genetic algorithm used to evolve the variations of fault specifications was implemented using the DEAP library [22]. The DEAP library includes various built-in genetic algorithms such as *eaSimple*, *eaMuPlusLambda*, *eaMuCommaLambda*, *eaGenerateUpdate*, but for the current usage a custom algorithm was implemented in order to have full control of the design. The genetic algorithm follows the basic workflow of an evolutionary algorithm including mutation, crossover, and selection processes. The first step is to initialise the population and calculate the initial fitness of each individual within the population. Then for every generation an offspring is created using roulette selection [23]. Each individual within the offspring is then mutated or mated based on a selected probability. Then the new fitness of the modified individuals is calculated using the defined evaluation function and the individuals are copied back to the initial population. The core components of the genetic algorithm are explained below:

a) Individuals

Each fault specification is represented by a list of 20 random selected Fault objects which are assigned random start and finish times. The individuals are randomly created from a pool of 30 type of faults, thus creating a number of random fault specification variations.

b) Mutation

A custom mutation function was implemented for the current use case which iterates over individuals and randomly modifies start and finish time as well as the value related to the specific type of fault. The mutation function ensures that start and finish times of faults that belong to the same class are not overlapped, to avoid the case where one fault disables the effect of another one which affects the same message, such as the pair of *ActivateSensor* and *DeactivateSensor* faults. The probability of mutation function is set to 0.7.

c) Crossover

For the crossover procedure, a custom-made function is implemented which ensures that the restriction of not having overlapped faults which affect the same messages is not violated. The function is following a similar approach of the one-point crossover [24] but is modified to exchange faults between two individuals in a way of retaining a valid sequence of faults based on the mentioned restriction. The maximum number of faults exchanged between individuals is equal to the half of their current size. The restriction will eventually create individuals which are not equally sized, but this is a trade-off which is tolerated for the sake of implementation. The probability of crossover function is set to 0.7.

d) Selection

The chosen selection method is the roulette method (selRoulette [25] in DEAP library) which selects k individuals from the input using k spins of a roulette.

e) Fitness function

Fitness functions is one of the most important parts of a genetic algorithm which highly affects evolution of individuals. The fitness function chosen for the single objective optimization [26] is:

$$f \ = totalGoalViolations \ x \ 0.3 + goalsNum \ x \ 0.7 \qquad (1)$$

and for the specific goal-focused case is:

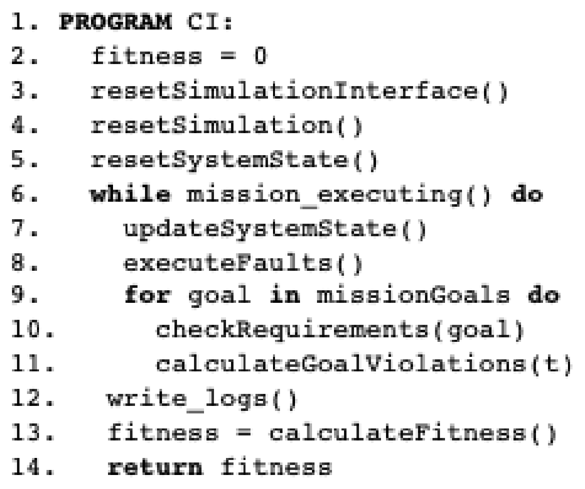
$$f = \sum_{1}^{n} totalGoal_{n}Violations \qquad (2)$$

The decision of choosing (1) is made in order to both prioritize a high number of distinct goals which are violated as well as aiming for a high number of total goal violations. This way the genetic algorithm will force evolution towards solutions which are characterized critical for a mission, and specifically fault specifications which will cause a high number of total goal violations between as many as possible distinct goals. The selection of (2) relies on benefitting fault specifications which will trigger a high number of violations on a selected set of n goals, forcing evolution towards a specific domain of goal violations. This will enable applying multi-objective optimisation [27] to the results in order to find the non-dominated solutions which maximize the multi-objective function (3), where $g_n(v)$ function returns the number of total violations on goal n. Both fitness functions will produce two different solution spaces, which are analyzed further on Section V *Experiments and Evaluation*.

$$\max\,(g_1(v), g_2(v), \ldots, g_n(v)) \qquad (3)$$

F. *Simulation Interface*

The simulation interface used for this implementation is written in Python using threads, and the roslibpy library. The simulation is generated from a model-to-text transformation using the user-defined mission model and acts as a communication bridge between the middleware and the running simulation. The simulation interface initialises the roslibpy topics for each robot property, for all robots defined in the mission model. These topics are generated using a model-to-text transformation which utilises the user-defined mission model that describes the specification for each robot. The interface utilises the publish-subscribe model provided by the roslibpy library through the concept of topics. Roslibpy takes advantage of the Rosbridge [28] component of ROS to establish a communication bridge between publishers and subscribers. When the interface starts running, it subscribes to all the generated topics, and each time there are new data published on a specific topic, a call-back function is triggered to store the data to a dictionary using a key value from where the key is the topic name. The data stored in the *topics* dictionary are then extracted by various *get* functions which are generated by the model-to-text transformation, based on the robot's specification and properties. This way of implementation enables the CI algorithm to easily consume the necessary data and properties of each robot, in order to continuously update its maintained local state of the mission, which will be then used at the decision-making procedure in order to maintain goal satisfaction.



```
PROGRAM CI:
  fitness = 0
  resetSimulationInterface()
  resetSimulation()
  resetSystemState()
  while mission_executing() do
    updateSystemState()
    executeFaults()
    for goal in missionGoals do
      checkRequirements(goal)
      calculateGoalViolations(t)
   write_logs()
   fitness = calculateFitness()
   return fitness
```

Fig. 4: CI Algorithm

G. *Collective Intelligence Algorithm*

Each mission is guided by a collective intelligence (CI) which runs on the controlling server and is responsible for the decision-making process of the multi-robot system, during mission execution. For the designed system, a CI template is generated through a model-to-text transformation which utilises the mission model and populates the template with template functions related to each individual goal, as well as all the necessary utility code needed for the algorithm to operate correctly. This creates high flexibility for the users, on implementing the logic of the CI, by letting the user to fill the generated function templates with hand crafted code, related to goal requirements and decision-making process according to their needs. The CI algorithm interacts with the simulation through the generated simulation interface, which utilises the publish-subscribe model provided by the roslibpy to enable exchange of information between the two endpoints. This way creates separation of concerns regarding the mapping between the high-level CI actions and the low-level actions performed by the running simulation, thus enabling ease of use of different custom-made CI designs, without affecting the interaction with the running simulation.

The workflow of the developed CI algorithm used on section V for experimental evaluation of the system's performance can be seen in Figure 4. The basic algorithm used, firstly resets the mission's state, a state which is maintained locally and keeps updating based on information provided by the robots' states within the running simulation. This local state it is used as a source of truth for the evaluation of the current state of the system within the simulation, in order to figure out which is the next best action to be performed, including actions such as increasing/decreasing speed, changing direction of movement etc according to a set of mission goals which can be seen on Table I. Then while the simulation is executing, the algorithm updates the current local maintained state of mission and the multi-robot system, using the current speed, energy, position, and any other robot information published in the ROS topics [18] inside the generated simulation interface, by the robots within the running simulation. After the local maintained state is updated, the algorithm executes all the faults which are associated with the current time interval. If the fault type is related to affecting messages exchanged between the server and components, the algorithm modifies the local maintained state of the mission based on the fault type, including altering speed, energy, coordinates, sensor information etc. in a way the executed fault defines, thus differentiating the local state from the original state within the simulation. If the fault type is about generating fake messages associated with unwanted actions such as start or stop a robot, activate, or deactivate a sensor, the algorithm sends these commands to the robots within the simulation through the simulation interface, which enables communication between the middleware and the running simulation. Then, the algorithm proceeds to check the requirements which satisfy all the defined mission goals, by using the filled with user-implemented logic function templates, in order to evaluate which goals are violated based on the current local state. According to the results of the checks, the CI performs the necessary actions in order to bring the system into an optimal state, in which the goals are not violated anymore. The algorithm also generates information related to the amount of total goal violations, goal violations per time interval and total violations per goal, which will be later used as metrics related to the evolution process of the population of fault specification variations, which are inputted to the genetic algorithm, in order to generate critical fault specification variations.

H. ROS log-based simulation

The simulation used for this implementation is a custom simulation implemented in Python using threads and the roslibpy library. The simulation is generated from a model-to-text transformation using the user-defined mission model. The simulation firstly initialises the robot objects with all their components and properties defined on the model, using all the necessary Python classes within the hand-crafted module which implements the mission DSL. Then, the MRS simulation sets the roslibpy topics for each robot property, for all robots within the simulation. These topics are generated using a model-to-text transformation which utilises the user-defined mission model that describes the specification for each robot.

When the simulation starts, for each time interval of the mission's duration, the robots execute their move functionality and they continuously publishing their current state (speed, position, energy, samples gathered etc.) to the appropriate roslibpy topics in order to be consumed by the CI algorithm through the simulation interface which will be used to update the local mission state of the offshore server. Furthermore, the simulation code includes some utility functionality generated using the model-to-text transformation. This functionality includes actions such as resetting and stopping simulation which is used during the iterative running of simulations during the evaluation of fault specification variations, as well as functionality for altering robot properties on demand, including changing direction, speed etc., when the CI sends the appropriate instruction. The simulation generates logs which track the current state of each robot, including current speed, position coordinates, energy etc. on each time interval, as well as various events occurred during the simulation. These events include changes on speed, direction of movement, activation/deactivation of components etc., which are instructed by the CI.

IV. PROTOTYPE

The developed prototype which will be used for experimental and evaluation purposes utilises a vast combination of technologies and techniques from the software engineering domain. There is an extensive usage of the Eclipse Epsilon language family for both mission definition and code generation. Each of the missions is defined through custom DSLs implemented in EMF. The backbone infrastructure such as CI template, middleware, simulation interface mission loader and fault specifications, is generated through a series of model-to-text transformations using a combination of EGL and EOL [29]. Hand crafted code is combined with the generated templates which implement the logic of the whole prototype. DSL classes and CI implementation, fault engine, simulation interface and custom simulation are all written in Python language. Furthermore, the prototype utilises the public subscribe model to establish communication between the middleware and MRS simulation. This is achieved through the use of ROS and specifically the use of roslibpy library, which is written in Python and its built-on top of the Rosbridge component of ROS. Rosbridge and roslibpy enable the communication between the two endpoints of middleware and MRS simulation utilising the publish-subscribe and topic concepts. The codebase of the prototype as well as installation instructions are available at https://github.com/tsokkisl/MSc-Project.

TABLE I. MISSION GOALS

ID	Description
g1	Avoid Collision: Each robot must avoid collision with other robots or obstacles.
g2	Stay Within Mission Area: All robots must operate within a defined mission area.
g3	Sufficient Energy: All robots should have sufficient energy to complete the mission.
g4-6	Gather Samples: All robots must gather samples from specific defined areas within mission area.

V. EXPERIMENTS & EVALUATION

A. Research Questions

The research questions which are aimed to be answered through the experimental evaluation are divided into two case studies, A and B. These two categories are A) single-objective optimization and B) multi-objective optimization using Pareto front for the analysis of the results, which are going to be investigated in order to check whether the research questions RQ 1-3 can be satisfied.

1) Case study A

RQ1 Can the system continuously produce through evolutionary computation, critical meaningful fault specifications? This research question is used to determine whether the system can produce critical fault specifications which can trigger a high number of goal violations on a broad set of different goals and can lead to catastrophic outcomes for the mission. These specifications can be analysed, aiming on prediction and proactiveness against critical fault occurrences.

RQ2 Can the system identify patterns of faults between solutions provided by the genetic algorithm, which contribute to a high number of goal violations? This research question is used to determine whether the system provides a way of identifying patterns between fault specifications, where certain faults can trigger a high number of goal violations. Identifying these kinds of patterns will help users to set a high priority on eliminating these specific faults.

2) Case Study B

RQ3 Can the system offer support on analysing the solution space and selecting solutions according to a specific ratio of violations between certain goals? This research question is used to determine whether the system can offer the ability to the users through analysing the solution space, to select fault specifications which are focused on a specific ratio of violations between a specific set of mission goals e.g., the user can choose a solution within the solution space which satisfy a specific ratio (2:1) of violations between goals *gx* and *gy*, where's *gx* is an energy-focused goal and *gy* is a collision-focused goal.

B. Experimental Methodology

1) Experimental Setup

The experimental procedure was performed on the mission specification, defined on the map in Figure 5. There are two generic categories of experiments. The first set of experiments comprised single-objective optimisation for the evolution of the fault specification variations. The second set of experiments was performed on a genetic algorithm focused on violations of a specific set of goals. Both set of experiments used the same CI algorithm and the same parameters of genetic algorithm, including mutation and crossover probabilities (0.7 and 0.7 respectively) and methods, selection method (roulette), a population of 20 individuals and 50 generations of evolution. Each individual represents a list of 20 faults generated through the fault specification loader. The only difference between the two variations of the genetic algorithm which were used for the experiments, was the chosen fitness function of the genetic algorithm. The experiments run on an Ubuntu virtual machine (VirtualBox 6.1) with 2Gb RAM and a 2 GHz Quad-Core Intel Core i5.

2) Case Studies

The developed system is evaluated through two 2 case studies. Specifically, the system is divided into two variations regarding of how the evolution is progressed, through the appropriate selection of fitness function. The first fitness function (1) is focused on single-objective optimization and specifically on benefiting fault specifications which combine two characteristics: i) violations on as many as different goals and ii) the highest number of total goal violations. The second fitness function (2) focuses on fault specifications which have high number of total goal specifications regarding specific goals chosen by the user and utilizes the multi-objective function (3) to analyze the solution space. Both case studies make use of the same CI shown in Figure 4 and explained on Section III as well as the same mission specification described by the model show in Figure 3. The defined prototype mission for this experimental evaluation is a simple underwater exploration mission in which the robots must enter a specific defined broad area with multiple obstacles scattered, and gather samples (water pressure, temperature, and depth) from distinct sub areas within the mission area, while avoiding any collision with the obstacles and the other robots, managing their energy resources, and staying within the appropriate areas within the mission area, during mission execution. A complete description of each of the prototype's mission goals can be seen in Table I. The responsibility of satisfying the mission's goals requirements is carried by the CI algorithm. The CI algorithm which runs on the offshore server, takes all the necessary actions including modifying robot speed, change direction of movement etc. in order to ensure that no mission goal is violated according to the current mission's state. The use of the same CI algorithm and mission specification is a deliberate choice, which enables system to deeply focus on fault injection and fault specification variations evaluation parameters, without having to consider the factor of different mission and CI variations when evaluating the impact of each fault specification. The mission is executed for *T = 1000* time-interval iterations, on an area of radius r = 2000 units, and as it was previously mentioned the genetic algorithm configuration consists of a starting population of 20 fault specifications and 50 generations of evolution. In the current mission, the robot team consist of 3 members with the specifications show on TABLE II.

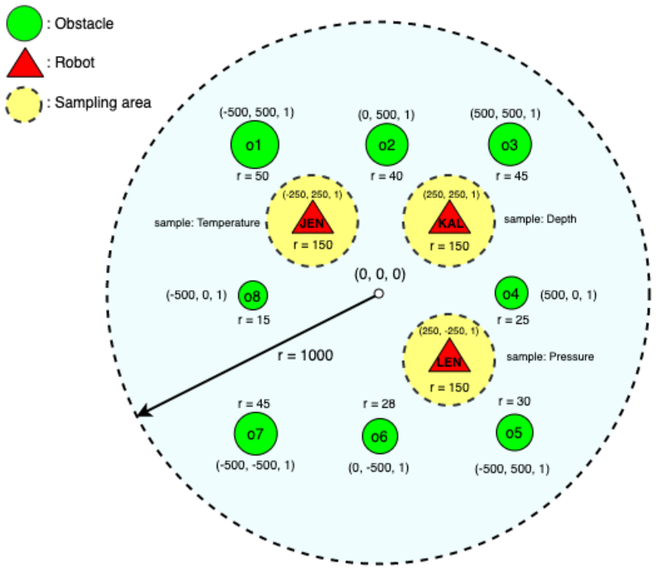


Fig. 5: Mission map

TABLE II.

Robot	Robot Specifications				
	Position	Speed	*Energy*	*Sensors*	
LEN	(250, -250, 1)	5	100000	GPS	Pressure
JEN	(-250, 250, 1)	4	64000	GPS	Temperature
KAL	(250, 250, 1)	4	100000	GPS	Depth

A representation of the mission's map with all the necessary information such as robot starting positions, obstacle positions, mission and goal areas can be seen in Figure 5. During mission execution, the CI consumes metrics published on simulation interface regarding multi-robot system's current state, in order to take actions which, ensure goal satisfaction. The simulation generates logs every interval which continuously capture the multi-robot system's state as well as events happened during the simulation, such as collisions or changes to the robots' parameters including speed, direction etc.

C. *Results and Analysis*

RQ1 This research question is related to the ability of the system to produce critical fault specifications where's critical corresponds to a) high number of goal violations and b) violations on a broad set of different goals. This research question falls into the case study of single-objective optimisation where the objective function is the fitness function (1) provided to the genetic algorithm. The way that fitness function is defined, it enables focusing on both a) and b) requirements with a certain incline towards requirement b) since a fault specification which affects a broader range of goals can be characterised as more interesting than a fault specification which just triggers high number of violations on a small number of distinct goals. The plots on Figure 6 and Figure 7, are produced from the metrics provided by the genetic algorithm. Figure 6 shows the mean fitness of fault specifications which is generated by injecting the fault specifications through the fault injection engine to the running simulation and evaluating and number of goal violations and the number of distinct violated goals according to (1), for every generation of the evolution. A similar plot shown in Figure 7, illustrates the fitness of the best fault specification produced, on each generation within the evolution procedure. As we can see from the two plots, there is a general increasing trend to both the mean fitness and the fitness of the best fault specification on each generation which indicates that the system is capable to produce critical fault specifications which satisfy both requirements a) and b) mentioned above. These observations can clearly indicate that the system is capable of satisfying research question RQ1, and that as long as the evolution is running for more and more generations, the genetic algorithm will tend to produce more and more critical fault specifications.

RQ2 This research questions investigates whether the system can identify certain patterns and fault repetitions related with the faults specifications which trigger high numbers of goal violations. Since the system can produce fault specifications which score high on goal violations according to RQ1, its relatively easy to analyse the produced logs and metrics to identify certain patterns in faults such as number of occurrences of specific faults in fault specifications produced by the genetic algorithm, which scored a high number of goal violations. This will enable users to identify those faults to evaluate their contribution to the amount of fault violations during mission execution, as well as focus on ways of mitigating these faults. Furthermore, in combination with RQ3 the users will be able to observe and map the occurrences of certain faults within the critical fault specifications, with violations of specific mission goals e.g., energy-wised, or collision-wised goals to identify the critical faults associated with specific goals. As can be seen in Figure 12, there is a representation of the number of occurrences of each of fault type within the top ten fault specifications in terms of goal violations. It is easy to distinguish that the top ten high scored fault specifications are highly influenced by the occurrence of specific types of faults such as:

FixedObstacleCoordinates:	19
DecrementRemainingEnergyCapacityReport:	10
ActivateSensor	10
FixedRobotCoordinates:	9
ZeroRemainingEnergyCapacityReport:	9
DecrementSpeed:	9

These observations can greatly benefit the design process of the system, to avoid future possible failures during mission execution.

RQ3 This research question investigates the ability of the designed system to offer support of extracting fault specifications, which satisfy a certain ratio of goal violations between a set of goals, through analysing the solution space provided by the genetic algorithm. This will give the flexibility of analysing fault specifications which focus on violations of a specific domain of goals, such as violations on energy-wised or collision-wised goals. In this case study, there is an extensive focus on goals specified by the user, thus the fitness function (2) of the genetic algorithm is different than the one from the single-objective optimisation. The fitness function provided to the algorithm, aims to benefit fault specifications which score high on violations of a set of goals, specified by the user. Similarly, from before, the plots on Figure 8 and Figure 9, illustrate metrics provided by the execution of the genetic algorithm, at the end of evolution cycle, which correspond to mean fitness and fitness of the best individual of each generation, respectively. As we can observe from the plots, the genetic algorithm can produce fault specifications which follow a steadily rising trend regarding fitness. This procedure will populate the solution space with fault specifications which score a high number of violations on the selected set of goals, which can be then used on the second part of analysis that aims on extracting the best solutions. Since the solution space is created, to be able to identify and analyse the interesting fault specifications which comprise different ratios of violations on the set of goals selected, we need to extract the non-dominated solutions from the solution space. This can be achieved through forming the Pareto front [30] in which those non-dominated solutions reside, as can be seen in Figure 10 and Figure 11. Figure 10 shows the non-dominated solutions of certain generations of evolution and Figure 11 illustrates all the non-dominated solutions from the whole solution space generated through evolution. Both plots refer to fault specifications which are focused on violations of goals *g1* and g3, and correspond into two different domains, since *g1* is a collision-focused goal and *g3* is an energy-focused goal. In this way, the users can choose to focus on the fault specifications which satisfy a certain criterion in respect of goal violation ratio on a selected set of goals, among a set of non-dominated solutions which reside

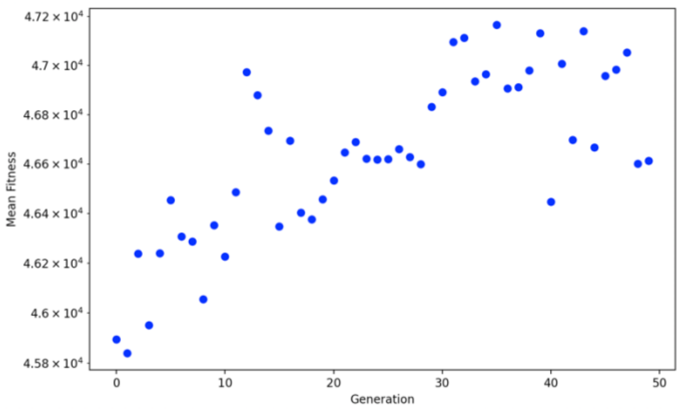


Fig. 6: Mean Fitness per each generation

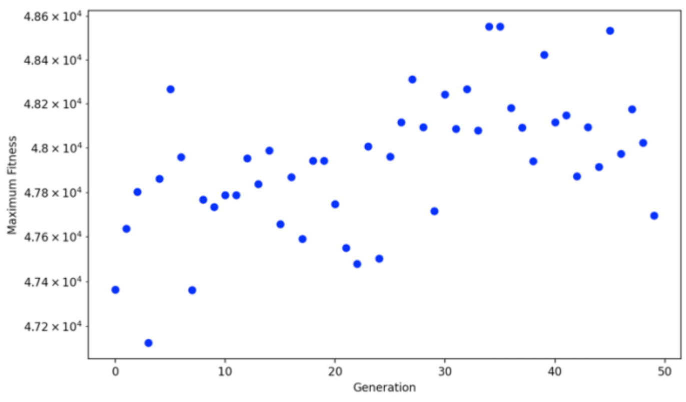


Fig. 7: Maximum fitness per generation

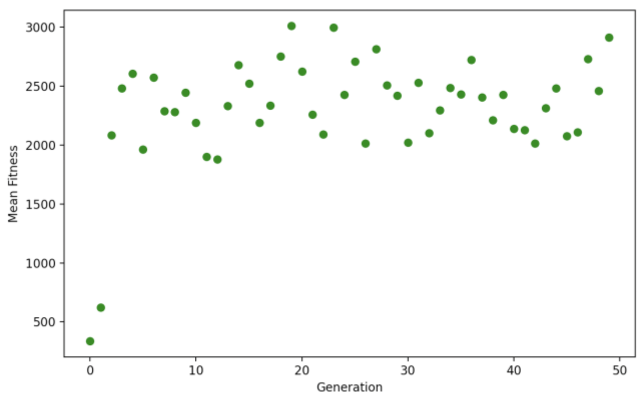


Fig. 8: Mean Fitness per each generation

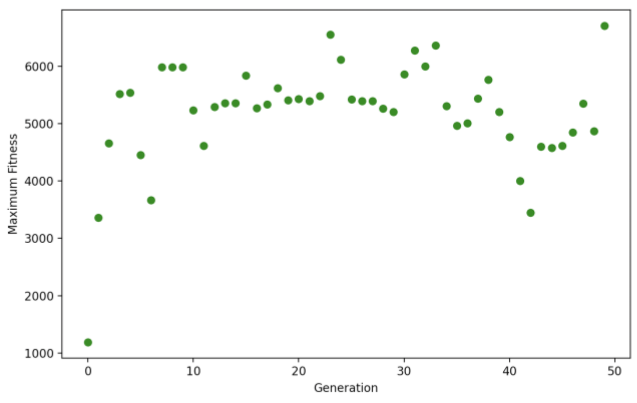


Fig. 9: Maximum Fitness per generation

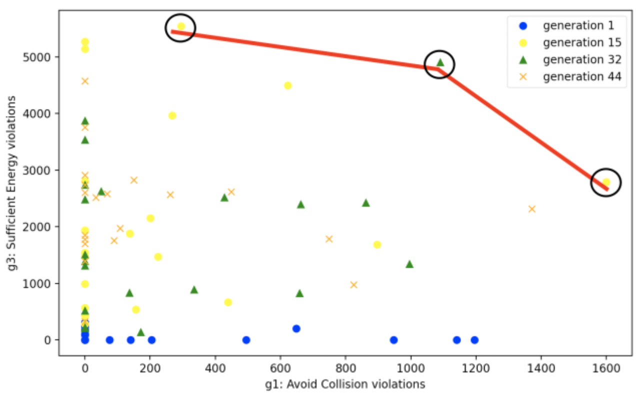


Fig. 10: Pareto front of *g1* and *g3* goal violations for randomly selected generations

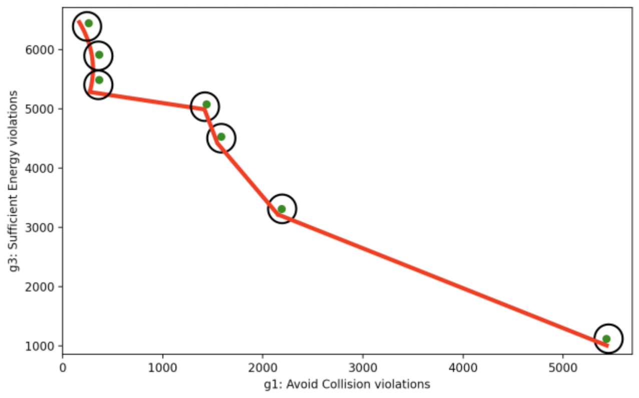


Fig. 11: Pareto front of all the non-dominated solutions of *g1* and *g3* goal violations

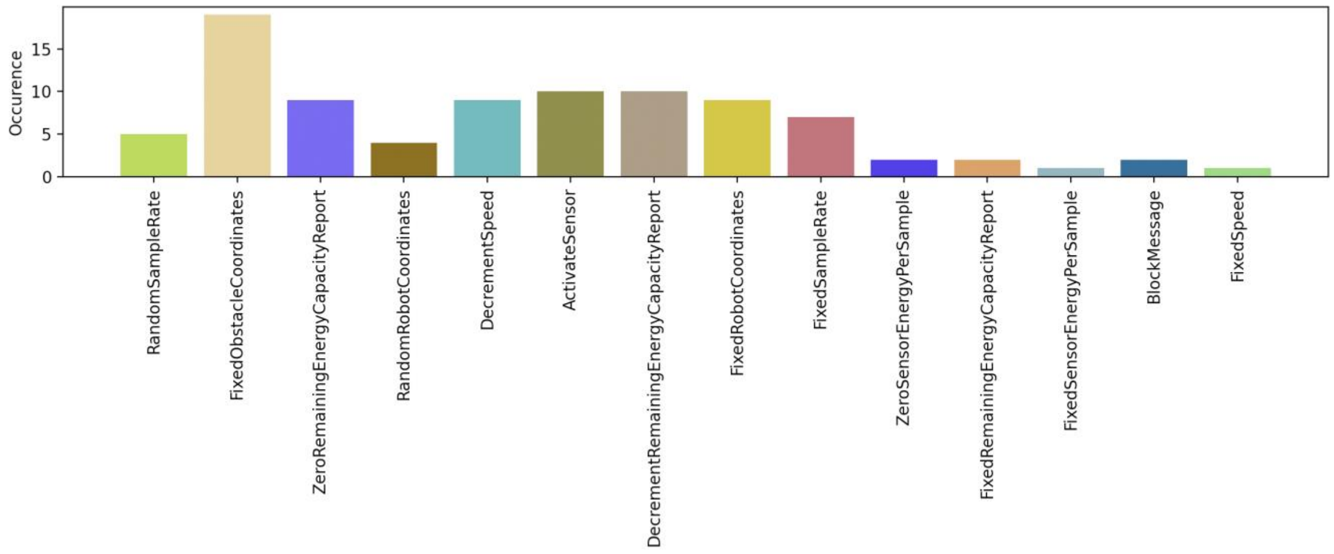


Fig.12: Occurrences of fault types for the top ten scoring fault specifications

on the Pareto front, generated using the multi-objective function (3). Specifically, in this experiment, the users can easily choose between fault specifications which focus more on violating energy restrictions (*g3*) or fault specifications which focus more on triggering a high number of collisions (*g1*). This observation can clearly indicate that the system is capable of satisfying RQ3 of the multi-objective optimisation case study, regarding flexibility of focusing on a specific domain of goal violations.

D. *Threads to Validity*

a) Mutation and Crossover constrains

The implementation of the mutation and crossover mechanisms is highly restricted by the requirement of not having time collisions between faults of the same class which could lead into mutual cancellation of faults, e.g., a time collision between the ActivateSensor and *DeactivateSensor* faults. This created a layer of restriction to the mutation and crossover mechanisms. Specifically, this influenced the design of the mutation method, in a way which is limited to modifications on the starting and finishing time, as well as to the value of the related to the fault class parameter, and it does not transform completely the fault into a different fault class. For the crossover function, due to the time collision restriction, the crossover is a custom-made crossover which attempts to simulate the one-point crossover by swapping faults from one individual to another, up to a maximum of the half length of the current individual which is not the usual type of crossover such as one-point or k-point crossover. These two restrictions to the mutation and crossover functions could affect the performance of the genetic algorithm. To mitigate this problem, some adjusting, and validation mechanisms are going to be developed as part of the future work described on Section VII.

b) Implementation – thread synchronisation

As part of the design and implementation of the whole system, the CI algorithm, simulation interface and the simulation itself are running on 3 different Python threads. The fact that these 3 processes which are running in parallel, are constantly exchanging data, creates a challenge, a challenge regarding synchronisation of the threads which is mandatory for the correct behaviour and operation of the system. Depending on the running platform and race condition [31] situations, the system might face synchronisation problems between those 3 running processes in terms of data exchange, which could slightly affect the accuracy of the system's operation.

c) Simulator

The current implementation comprises a simple custom-made log-based simulator, which is built on top of the roslibpy and utilises the Rosbridge component of the ROS as well as the publish-subscribe model to support communication between the components of the multi-robot system. This creates a limitation problem for both supported functionality and visualisation. The custom simulator stores the current state of the multi-robot system and the events occurred into logs, and provides no visualisation options for the current state of the mission rather reading from the logs, as well as no additional functionality to the user, in contrast to other complete simulators such as Gazebo, UNDERSEA [32] etc. To mitigate these limitations, the system could be modified to support the mentioned simulators as part of the future work described on Section VII.

d) Mission specification

During the experiments and evaluation procedure, there was only one mission specification which was used for the two case studies. This creates the need of performing more experiments and evaluations, on different variations of missions and case studies, in order to assume generalization of the acquired results into different case studies. This is planned to be a part of the future work which is described on Section VII.

VI. Conclusion

This paper introduced an evolution-based fault injection method for multi-robot systems, through the design and implementation of a system which comprise the model-driven engineering approach. There was an extensive description of the system's architecture including analysis of all the fundamental components which are involved during the system's operation. The proposed system was evaluated through a developed prototype, by using two case studies, one for single-objective optimisation analysis and one for multi-objective optimisation analysis of the results. The results of evaluation clearly indicated that the system is capable of identifying critical fault specifications which can lead into both a high number of goal violations as well as violations on a high number of distinct goals, through genetic evolution techniques. Furthermore, the system was able to provide the capability of producing fault specifications which focused on violation of a user-specific defined set of goals through multi-objective optimisation (3). Lastly, the system was able to identify patterns and repetitions of faults within the fault specification variations, which are contributing to high numbers of goal violations. Generally, the system proved promising on serving the general purpose of identifying critical fault scenarios which could lead to unwanted and disastrous mission outcomes, to promote proactiveness against such scenarios, as well as mitigation of such faults.

VII. Future Work

As it is mentioned on the *Threads to Validity* Section, in the future, the system is planned to extend its flexibility and functionality by enabling support of popular simulators such as Gazebo, UNDERSEA etc., In addition, it is planned to develop adjusting, and validation mechanisms for the mutation and crossover functions of the genetic algorithm, in order to relief from the time-collision restrictions which limit the capabilities of those two functions and lastly, perform experiments on different mission and case studies variations, in order to be able to safely assume generalization of the acquired results. Lastly, more work can be done on evaluating the system, with the use of other sophisticated evolutionary algorithms provided by the DEAP library such as *MuCommaLambda* [33], *MuPlusLambda* [33], GenerateUpdate [33], as well as experiment on a range of genetic algorithm parameters such as mutation and crossover probabilities and methods, selection, and fitness functions etc.

References

[1] D.S. Kolovos, R.F. Paige, AND F.A.C Polack F.A.C., "The Epsilon Transformation Language," *2008*, In: Vallecillo A., Gray J., Pierantonio A. (eds) Theory and Practice of Model Transformations. ICMT 2008. Lecture Notes in Computer Science, vol 5063. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-69927-9_4

[2] K. Sastry, D. Goldberg and G. Kendall, "*Genetic Algorithms,*" In: Burke E.K., Kendall G. (eds) Search Methodologies. Springer, Boston, MA, 2005, https://doi.org/10.1007/0-387-28356-0_4

[3] Eclipse, *The Epsilon Generation Langiage (EGL)*, [Online]. Available: https://www.eclipse.org/epsilon/doc/egl/

[4] A. Favier, A. Messioux, J. Guiochet, J. Fabre and C. Lesire, "A hierarchical fault tolerant architecture for an autonomous robot," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, Valencia, Spain, 2020 pp. 122-129. doi: 10.1109/DSN-W50199.2020.00031 keywords: {fault tolerance;fault tolerant systems;fault trees;robot kinematics;osmosis;robot sensing systems} url: https://doi.ieeecomputersociety.org/10.1109/DSN-W50199.2020.00031

[5] Gazebo, Gazebo *Robot simulation made easy,* [Online]. Available: https://www.ros.org/about-ros/

[6] D.Garrido, L Ferreira., J.Jacob and D.C. Silva, "Fault Injection, Detection and Treatment in Simulated Autonomous Vehicles," In: Krzhizhanovskaya V. et al. (eds) Computational Science – ICCS 2020. ICCS 2020. Lecture Notes in Computer Science, vol 12137. Springer, Cham. https://doi.org/10.1007/978-3-030-50371-0_35

[7] Microsoft, Microsoft Flight Simulator, [Online]. Available: https://www.xbox.com/en-US/games/microsoft-flight-simulator

[8] G. Juez Uriagereka *et al*., "Design-Time Safety Assessment of Robotic Systems Using Fault Injection Simulation in a Model-Driven Approach," *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, 2019, pp. 577-586, doi: 10.1109/MODELS-C.2019.00088.

[9] F. Wotawa, "Testing Self-Adaptive Systems Using Fault Injection and Combinatorial Testing," *2016 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, 2016, pp. 305-310, doi: 10.1109/QRS-C.2016.47.

[10] S. Jha, S. Banerjee, T. Tsai, S. Hari, M. Sullivan, S. Keckler, Z. Kalbarczyk and R. Iyer, "ML-Based Fault Injection for Autonomous Vehicles: A Case for Bayesian Fault Injection," *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 10.1109/DSN.2019.00025.

[11] Christensen, A.L., O'Grady, R., Birattari, M. et al. Fault detection in autonomous robots based on fault injection and learning. Auton Robot 24, 49–67 (2008). https://doi.org/10.1007/s10514-007-9060-9

[12] L. Hardesty, *Explained: Neural networks*, MIT News Office, April 14, 2017. [Online]. Available: https://news.mit.edu/2017/explained-neural-networks-deep-learning-0414

[13] Y. Fu, A. Terechko, T. Bijlsma, P. J. L. Cuijpers, J. Redegeld and A. O. Örs, "A Retargetable Fault Injection Framework for Safety Validation of Autonomous Vehicles," *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*, 2019, pp. 69-76, doi: 10.1109/ICSA-C.2019.00020.

[14] G. Juez, E. Amparan, R. Lattarulo, A. Ruiz, J. Pérez and Espinoza, E. Huáscar, "Early Safety Assessment of Automotive Systems Using Sabotage Simulation-Based Fault Injection Framework," *2017,* 255-269. 10.1007/978-3-319-66266-4_17.

[15] Dynacar, Dynacar by technalia, [Online]. Available: http://dynacar.es/en/home.php

[16] D. Kolovos, L. Rose, R. Paige, and A. Garcıa-Domınguez, "The epsilonbook,"Structure, vol. 178, pp. 1–10, 2010.

[17] ROS.org, *About ROS,* [Online]. Available: https://www.ros.org/about-ros/

[18] Roslibpy Python ROS Bridge library, *Examples,* [Online]. Available: https://roslibpy.readthedocs.io/en/latest/examples.html

[19] Eclipse, *Epsilon and EMF,* [Online]. Available https://www.eclipse.org/epsilon/doc/articles/epsilon-emf/

[20] www.moos-ivp.corg, *MOOS-IvP*, [Online]. Available: https://oceanai.mit.edu/moos-ivp/pmwiki/pmwiki.php

[21] IBM, *Publish/subscribe overview*, Accessed on: August 24, 2021. [Online]. Available: https://www.ibm.com/docs/en/app-connect/11.0.0?topic=applications-publishsubscribe-overview

[22] DEAP, *DEAP Documentation,* [Online]. Available: https://deap.readthedocs.io/en/master/

[23] Newcastle University, *Roulette wheel selection*, [Online]. Available: http://www.edc.ncl.ac.uk/highlight/rhjanuary2007g02.php/

[24] GeeksforGeeks, *Crossover in Genetic Algorithm,* [Online]. Available: https://www.geeksforgeeks.org/crossover-in-genetic-algorithm/

[25] DEAP, *Evolutionary Tools,* [Online]. Available: https://deap.readthedocs.io/en/master/api/tools.html

[26] D. Savic, "Single-objective vs. multiobjective optimisation for integrated decision support," *2002 Proceedings of the First Biennial Meeting of the International Environmental Modelling and Software Society*, vol.1, 2002, pp.7-12.

[27] K-H. Chang, "Multiobjective Optimization and Advanced Topics," *2015,* 10.1016/B978-0-12-398512-5.00005-0.

[28] ROS.org, *rosbridge_suite,* [Online]. Available: http://wiki.ros.org/rosbridge_suite

[29] Eclipse, *The Epsilon Object Language (EOL),* [Online]. Available: https://www.eclipse.org/epsilon/doc/eol/

[30] A. Lavin, "A Pareto Front-Based Multiobjective Path Planning Algorithm", 2015

[31] Baeldung, *What is Race Condition*, Accessed on: October 19, 2020. [Online]. Available: https://www.baeldung.com/cs/race-conditions

[32] S. Gerasimou, R. Calinescu, S. Shevtsov and D. Weyns, "UNDERSEA: An Exemplar for Engineering Self-Adaptive Unmanned Underwater Vehicles," *2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, 2017, pp. 83-89, doi: 10.1109/SEAMS.2017.19.

[33] DEAP, *Algorithms,* [Online]. Available: https://deap.readthedocs.io/en/master/api/tools.html