SESAME
SECURE AND SAFE MULTI-ROBOT SYSTEMS

Project Number 101017258

D6.2 Simulation-Based Testing Methodology for EDDIs

Version 1.0
30 June 2022
Final

Public Distribution

University of York

Project Partners: **Aero41, ATB, AVL, Bonn-Rhein-Sieg University, Cyprus Civil Defence, Domaine Kox, FORTH, Fraunhofer IESE, KIOS, KUKA Assembly & Test, Locomotec, Luxsense, The Open Group, Technology Transfer Systems, University of Hull, University of Luxembourg, University of York**

Project Partner Contact Information

Aero41 Frédéric Hemmeler Chemin de Mornex 3 1003 Lausanne Switzerland E-mail: frederic.hemmeler@aero41.ch	**ATB** Sebastian Scholze Wiener Strasse 1 28359 Bremen Germany E-mail: scholze@atb-bremen.de
AVL Martin Weinzerl Hans-List-Platz 1 8020 Graz Austria E-mail: martin.weinzerl@avl.com	**Bonn-Rhein-Sieg University** Nico Hochgeschwender Grantham-Allee 20 53757 Sankt Augustin Germany E-mail: nico.hochgeschwender@h-brs.de
Cyprus Civil Defence Eftychia Stokkou Cyprus Ministry of Interior 1453 Lefkosia Cyprus E-mail: estokkou@cd.moi.gov.cy	**Domaine Kox** Corinne Kox 6 Rue des Prés 5561 Remich Luxembourg E-mail: corinne@domainekox.lu
FORTH Sotiris Ioannidis N Plastira Str 100 70013 Heraklion Greece E-mail: sotiris@ics.forth.gr	**Fraunhofer IESE** Daniel Schneider Fraunhofer-Platz 1 67663 Kaiserslautern Germany E-mail: daniel.schneider@iese.fraunhofer.de
KIOS Maria Michael 1 Panepistimiou Avenue 2109 Aglatzia, Nicosia Cyprus E-mail: mmichael@ucy.ac.cy	**KUKA Assembly & Test** Michael Laackmann Uhthoffstrasse 1 28757 Bremen Germany E-mail: michael.laackmann@kuka.com
Locomotec Sebastian Blumenthal Bergiusstrasse 15 86199 Augsburg Germany E-mail: blumenthal@locomotec.com	**Luxsense** Gilles Rock 85-87 Parc d'Activités 8303 Luxembourg Luxembourg E-mail: gilles.rock@luxsense.lu
The Open Group Scott Hansen Rond Point Schuman 6, 5th Floor 1040 Brussels Belgium E-mail: s.hansen@opengroup.org	**Technology Transfer Systems** Paolo Pedrazzoli Via Francesco d'Ovidio, 3 20131 Milano Italy E-mail: pedrazzoli@ttsnetwork.com
University of Hull Yiannis Papadopoulos Cottingham Road Hull HU6 7TQ United Kingdom E-mail: y.i.papadopoulos@hull.ac.uk	**University of Luxembourg** Miguel Olivares Mendez 2 Avenue de l'Universite 4365 Esch-sur-Alzette Luxembourg E-mail: miguel.olivaresmendez@uni.lu
University of York Simos Gerasimou & Nicholas Matragkas Deramore Lane York YO10 5GH United Kingdom E-mail: simos.gerasimou@york.ac.uk nicholas.matragkas@york.ac.uk	

Document Control

Version	Status	Date
0.1	Document outline	15 May 2022
0.4	First draft	15 June 2022
0.5	First full draft	18 June 2022
0.6	Revised first draft	22 June 2022
0.8	Revision after review	28 June 2022
1.0	Final version	30 June 2022

Table of Contents

List of Figures

Executive Summary

This document presents the first iteration of the simulation-based testing component of the SESAME platform. It focuses on the algorithm and tool developed for Task 6.2:

Task 6.2: Simulation-Based Testing Methodology for EDDIs

The document provides information on the specific challenges tackled in the implementation of simulation-based testing, identifying certain gaps in the literature which we address with our simulation-based testing tool. The approach we specify for simulation-based testing in the wider context of the SESAME project is presented, and implementation details are specified. An example of its application to a ROS-based case study is also presented.

The list of requirements emerging from SESAME partners that are addressed in our simulation-based testing framework are also presented in this document.

A methodology for simulation-based testing is presented based on the utilisation of a domain-specific language (DSL) in order to model the space of potential fuzz testing operations upon the multi-robot system (MRS). We utilise a DSL in order to allow non-developers to define the space of operations, raising the level of abstraction and providing simulator independence. This DSL is paired with code generation using standard Eclipse-based technologies, in order to produce middleware test runners that can interface with an MRS simulation and dynamically alter or disrupt its messages at runtime. We provide a flexible and generic interface for abstracting away specific details of simulator connectivity, providing extensibility to new simulators.

Our tool incorporates an evolutionary experiment runner which consumes experimental test campaign definitions, and defines and explores new tests dynamically, constrained by the testing space specified in the testing model. Scenario-specific performance metrics, which the user can define according to the violations they would like to discover, are used to obtain feedback as to the utility of test configurations discovered, and used as feedback to guide a multi-objective optimisation loop towards the most interesting regions of the testing space.

An application of our methodology is illustrated upon a ROS-based MRS healthcare case study, which is directly transferable to the ROS-based SESAME use cases such as the Locomotec hospital disinfection use case. An in-progress interface to other simulators, such as TTS for the KUKA use case, is also considered, showing how our generic interfacing methodology can be easily adapted to this use case.

Structure of the Document

- Section 1 provides an introduction to the challenges of simulation-based testing and how our approach aims to address these challenges
- Section 2 considers our simulation-based testing approach in the context of related work
- Section 3 presents in detail the methodology and architecture of our simulation-based testing approach
- Section 4 presents specific details about its technical implementation
- Section 5 presents the application of our tool to a ROS-based case study example
- Section 6 concludes the report

1 Introduction

1.1 Overview

This document summarises the activities concerning the simulation-based testing component aspect of the project. The work has focused on the development of a methodology and architecture for simulation-based testing for the SESAME platform.

This deliverable reports on the work carried out within Task 6.2, aiming to describe the technical components developed for simulation-based testing in work package WP6. To this end, we have identified all the components developed and requirements that have been satisfied for enabling their integration with other SESAME components.

In particular, we present a methodology for simulation-based testing, supported by a DSL to specify the fuzzing space and experiments at a high-level of abstraction. This is accompanied by a code-generation engine and an evolutionary experiment runner for dynamically generating and exploring a fuzz testing space.

In the following sections, we first motivate the development of our approach by considering the specific challenges presented in the area of simulation-based testing and shared by our use case partners, and then describe how our approach will contribute to solving these challenges.

1.2 Simulation-Based Testing

Simulation-based testing enables investigating the capacity of a multi-robot system (MRS) to operate dependably using a virtual environment, whose fidelity could range from low to very high, and which closely resembles the target deployment environment [62, 29, 10, 7, 60].

The preference for simulation-based testing is partly due to the high maturity level of modern robotic frameworks (e.g., ROS [52], MOOS-IvP [8] and simulators such as Gazebo [27] and Unity [38]) that enable realistic simulation of robots and their behaviour using rich navigation and mission planning software functions. Simulations can be performed at varying levels of physical realism, allowing users to explore trade-offs between performance/feature availability and realism. These functions can later be deployed with minimum changes and modest effort on real robots [62].

Also, simulation-based testing supports searching large design spaces of components and system configurations, using limited physical hardware resources, thus reducing significantly the overheads for detecting potential violations of safety requirements [33, 28]. Performing such tests in the real world using the physical MRS can present a number of challenges. Firstly, testing robotic systems in the real world is time-consuming and expensive [70]. Robots are typically procured only when companies are confident that the system configurations and the robot specifications are the desired and fulfil the system's goals. As a result, real world testing cannot happen in the early phases of the engineering lifecycle, which does not give the opportunity to developers to get early feedback on the proposed algorithm and configurations. Therefore, simulation-based testing can provide a viable approach to reducing the area of the design space that must be explored later with real-world testing.

The temporal dimension can be an important factor in robotic systems testing. There are cases and events that can only be evaluated after a specific amount of time has passed or other prerequisites have taken place. For example, assessing the safety of an autonomous braking system under heavy rain conditions in the real world is only possible after heavy rain occurs. Similarly, if a robotic arm can reach high temperatures, close to its maximum working limits after a few hours, which would require a specific cooling period. This challenge becomes even harder to overcome if the objective is to test and reproduce long-lasting missions or rare events.

In addition, testing of heterogeneous system configurations (e.g., different team compositions, different types of components, or components with different characteristics) is a complex and challenging activity. Furthermore, the scalability of MRS scenarios is another concern, as behaviour may change as scenarios scale up in

the real world. A related challenge involves testing corner cases or black-swan scenarios (in which rare or unexpected events lead to major and unanticipated MRS failures) in the real world. It is difficult and expensive, if at all possible, to assess specific corner cases that are rare and unexpected to happen frequently in the real world.

Therefore, simulation-based testing provides a viable route to assuring the quality of MRS and the correct behaviour of their components, especially in safety-critical systems. EDDI-related components should be able to monitor a robotic system, identify system faults, cyber-attacks or unexpected environmental conditions, and potentially trigger adaptations and behaviours to enable the system to cope with any problems. Given the important role of the EDDIs to support the correct and robust behaviour of MRS, it is important to test and assess their quality and capabilities before they are deployed in the real world.

1.2.1 Industrial Challenges

Although simulation-based testing is a useful approach for assessing the quality of MRS, there are significant challenges that may impact its effectiveness, when viewed in conjunction with real industrial case studies. The need to cover a wide variety of industrial sectors (reflected also by the project use cases) implies that simulated MRS systems can be significantly different in terms of physical features and in terms of control strategies. From the point of view of the testing infrastructure, this represents a further level of complexity.

Even when common simulation frameworks and standardised robotic middlewares are used, different assumptions may exist in how they are configured which impacts simulation structure. For example, in a distributed UAV system, ROS simulations may consist of multiple ROS master nodes which could require distinct simulation connections for interfacing (e.g., as in the case of the Cyprus Civil Defence - KIOS use case). These differences in the topology and methods of data storage may impact the testing strategies employed. Industrial simulations may consist of proprietary components to which interfacing for simulation-based testing may not be easy to achieve. Alternatively, connections to a simulation may require complex interfacing techniques, such as accessing shared memory, which can involve locking or other protocols that themselves subtly impact the behaviour of the MRS under test. Therefore, interfacing to real systems such as the ones used within the used cases of the SESAME project can be challenging.

Moreover, the validation of an EDDI-enabled MRS is based on the fulfilment of acceptance criteria that, at simulation level, must be mapped onto tasks monitoring the evolution of some variables of interest of the system over time and evaluating (possibly complex and articulated) expressions. Nevertheless, each simulation software implements its own events timeline and exposes a proprietary runtime information model to access and inspect the internal states of the environment.

Finally, quality assurance of functional and non-functional requirements is a challenge itself. Testing of non-functional requirements like reliability, safety and security is a non-trivial task. These activities require the definition of complex scenarios in which trade-offs between multiple and likely conflicting functional and non-functional requirements should be taken into account. These include not only intuitive cases, like sacrificing performance to achieve better security, but also counter-intuitive cases too. Consider for example the implementation of a door locking system which requires specific doors to be locked at all times for the system to be secure, but also requires the same doors to be accessible in the case of a fire, to assure the safety of the residents.

1.2.2 Modelling Challenges

The wide multiplicity of simulation software platforms available provides another related challenge. Each platform, in fact, is based on specific modelling formats and languages that are used to implement not only the static assets but also the executable part of the scenarios that cannot be represented using declarative approaches. Further, each simulation software implements its own events timeline and exposes a proprietary

runtime information model to access and inspect the internal states of the environment, meaning that an additional transformation layer must be considered to export test results back to the testing framework.

Besides the low-level technical challenges, in the developed system it should be considered that the final end-user of the envisioned simulation based testing platform should be the MRS developer with likely expertise in robot mission modelling, control automation and robot programming but unlikely to be an expert in simulation development. Therefore, the implemented solution should present the option of hiding as much of this complexity as possible, simplifying the authoring and execution of simulation experiments, and enhancing the comprehension of the working conditions that triggered a specific (correct or buggy) response of the MRS. In particular, this latter aspect requires that the simulation tools and their SESAME enabled interfaces support full traceability of data that determines the production of mission safety violations.

While at design time the concerns are mainly related to the model generation, at deployment time the key challenges are related to instrumenting a digital twin paradigm by propagating feedback from the real to the simulated system. The feedback is represented by a set of data streams, flowing from the operating MRS to the monitoring infrastructure, that must be transformed into meaningful updates of the simulated system behaviours. The first difficulty is related to the capability of capturing relevant modifications of system parameters that affect the coordinated logics of the robot fleet starting from raw values. The second challenge, as in the engineering phase, is related to the heterogeneity of the observed system variables, their differences in semantic, read/transfer rates and the nature of the digital model as well as to the differences of the specific simulation platforms. Finally, it is important to consider that this real-to-digital alignment must be automatized but should not be automatic, meaning that the end-user (system engineers) must be kept in the loop with dedicated supporting interfaces simplifying the task of evaluating and the automated analyses and the suggested modifications and applying and deciding whether to apply them to the current simulation model or to test variants.

1.2.3 Reproducibility Challenges

The unreliability of the components of robotic systems [1] and the emergent behaviour of those systems when interconnected at scale is also a significant challenge that should be taken into account [23]. Deploying robots that must follow strict requirements in terms of energy consumption, size, cost, etc. comes with the trade-off of potentially reduced reliability of their components (e.g., sensors).

It was also important to pay attention to the repeatability of the simulation-based testing experiments. Repeatability problems will become obvious if there are any intermittently non-deterministic components in the simulator, or the simulated system. For instance, in our past experience of MRS testing, the UAV planner produced slightly different trajectories at run time between different simulation runs [61].

In robotic testing it may be difficult to reproduce precisely a failing configuration on a different host simulation machine. These issues may arise due to system timing variations, which may lead, for example, to systems displaying wildly different behaviours when are subjected to transient faults or intentional fuzzing operations. Even with the same MRS test configuration, and precisely defined simulation platform versions, system libraries and parameters, random variations in precise mission start up location and velocity can lead to a different outcome if robotic behaviour has emergent characteristics. In several occasions, results produced in simulation-based testing, such as the behaviour of a UAV in a test scenario resulting in controller velocity errors, have produced variable output trajectories and therefore different potentials for collisions [61] In order to support replication is it important to record full logs from tools such as rosbags in ROS to assist reconstruction of the behaviour observed in a failing case. Although these do not provide output statistics, they can be helpful to replay the behaviour of the failing case.

1.2.4 Challenges for SESAME Use Cases

In this section, we report simulation-based testing challenges from the SESAME project use cases detailed in SESAME Deliverable D1.2 [48], refined following recent discussions with use case partners:

UVC Disinfection Use Case (Locomotec)
One of the challenges of this use case is accurately modelling the impact of the UVC lamps, and assessing the impact of the UV disinfection upon the particular surface for the mission goals. It is necessary to come up with a useful performance metric that quantifies the disinfection performance of a particular testing scenario.

This use case also has multiple implementations of the simulation available, using both Gazebo and Unity. Although both of these are capable of interfacing to ROS, this may potentially lead to divergence of simulation behaviour between both implementations under the same test configuration. In addition, there are differences in functionality, in that only the Unity implementation currently has direct support for human avatars. This is due to differences in physical realism between simulators, given that the greatest implementation effort was directed to Unity given its improved support for realistic rendering and physics.

Power Station Inspection Use Case (KIOS and Cyprus Civil Defence)
One of the challenging aspects of this use case is the environmental conditions, i.e., in the expected deployment case, there may be high temperature, strong winds, electromagnetic interference or radiation when being close to a power station. This may alter the behaviour of the nodes in real deployment, causing message losses and transient disconnection of nodes. Also, it may create a larger reality gap between simulation and the real deployment. This use case exhibits a distributed architecture with multiple ROS masters in the system, upon the ground control station and upon multiple vehicles. This may lead to complexities in interfacing with and specialising the testing platform for this use case.

Autonomous Pest Management in Viticulture (Domaine Kox, AERO41 and LuxSense
The weather is a challenge for the deployment of this use case. As the MRS are deployed in outdoor vineyards, procedures need to be independent to varying meteorological conditions (e.g., high changes in temperature and pressure, dust, precipitation). High wind speeds cause risks of UAV loss or damage. Simulators must also be capable of accurately simulating various relevant scenarios such as the presence of strong wind. They must also be capable of accurate detection of blocking objects in the area, and simulating an avoidance strategy.

Security Management of MRS-Base Assembly Lines (KUKA)
The architecture of this use case is relatively complex with the intersection of the simulator from TTS and SIMIT. These different simulation platforms are communicating over shared memory which may be difficult to interface with properly for simulation-based testing, without the addition of locking or other protocols that may interface with the testing platform. A custom interface to the shared memory will have to be developed, that can also interface with the TTS simulator.

1.3 Response To Challenges

In Section 1.2, a number of challenges were identified for MRS systems in relation to simulation-based testing. This section presents how the SESAME simulation-based testing architecture presented in this deliverable report contributes to addressing these challenges

1.3.1 Very large space of potential testing configurations

There is a very large space of potential simulation-based testing configurations, resulting from the large number of system components, potential failures that may affect them, and variations in their impact or intensity. In addition, these events can be transient and can occur in combinations. In order to make this vast space viable for real experiments, subsets of it have to be searched for evaluation, incorporating the expertise of system testing engineers and knowledge of potential attacks and security threats. The solution proposed in this report is firstly

to use model-driven engineering (MDE) and define a domain-specific language (DSL) to allow test engineers to specify searchable regions of this testing space. Our DSL will allow the specification of both the boundaries of the testing space and also the selection of subsets of this for particular experiments. Within these selected spaces, we incorporate an intelligent search strategy for experiments, using a genetic algorithm which constructs simulation tests dynamically, taking into account feedback from scenario-specific performance metrics. This allows automatic exploration of the selected testing space region, discovering configurations producing violations of system safety requirements.

1.3.2 Heterogeneous components

Our approach provides a way for these components to be modelled, and for generic operations for testing these components to be specified, through a hierarchy of available fuzzing operations. Fuzzing operations can be specified in a class hierarchy, which users can extend and add additional information to provide customised tests upon these components. Heterogeneous components should communicate where possible using standard message types and industry-standard messaging frameworks, so they are more easily composable.

1.3.3 Diversity of simulators provided and required

The large number of simulators available, and their underlying assumptions, data storage mechanisms and message formats, presents a problem for engineering a flexible testing framework. This provides a motivation for specifying a generic simulator interface, with specific simulator implementations, which we provide with our architecture in Figure 3. This allows users to select a particular simulator interface, and if necessary to customise the simulator interface to their needs. Using general-purpose libraries for interconnection and inter-process communication, e.g gRPC [31], provides increased flexibility for repurposing interfacing components when required.

1.3.4 Potentially complex simulation configurations

In situations in which there is a non-standard simulation configuration, code generation could provide the required flexibility, allowing a simulation-based testing platform to be configured to make multiple simulator connections. If the simulator interface is sufficiently generic, customised simulator interfaces could potentially be developed to make connections to multiple publish-subscribe databases simultaneously. Logical interface components on the simulator side can be used to, for instance, hide the complexity of shared memory access to simulator state.

1.3.5 Non-determinism and reproducibility

One of the key factors leading to non-determinism in simulation-based testing is related to timing offsets between different runs of the same configuration. In our previous research, we were able to compensate for the start-up time variation in real experiments by using a custom time reference, triggered only when the mission was ready to begin [61]. This also provides a viable strategy for situations in which there is variable startup delay of simulation processes. Therefore, supporting a flexible simulator interface, which allows custom time references for a particular simulation to be specified, can allow such timing stabilisation to be achieved.

Another approach which can be performed is to analyse the determinism of experiments, by supporting multiple executions of the same configuration, and to use search algorithms focused upon isolating the rare or unstable events. A further approach is to trigger fuzzing events in simulation-based testing based upon timing-independent conditions, for example based upon simulation state. This would reduce our dependency on precise timing and increase the platform's applicability to less deterministic missions that support significant variability in behaviour in multiple runs of the same configuration.

2 Related Work

2.1 Simulation-Based Testing

Simulations are used extensively in the engineering of robotic systems. The proposed work lies at the intersection of model-driven engineering, automated testing, and robotics.

Simulation-based testing has been used successfully in the AirSim simulation framework [58] which provides the ability to rapidly investigate a large number of potential configurations for drones and autonomous vehicles. Recent research has shown that real robotic system bugs may be detected by simulation-based testing and that a majority of real-world bugs may be investigated in simulation [62, 60].

One of the main purposes of simulation for robotics is to provide a safe and fully controlled virtual testing and verification environment. Afzal et al. [2] proposes a framework that can facilitate automated testing of robotic systems using software-in-the-loop (low-fidelity) simulations and anomaly detection. Similarly, Huck et al. [37] focus on testing industrial human-robot collaborative systems by using a human model and an optimization algorithm to generate high-risk human behaviour in simulation, thereby exposing potential hazards.

Simulation is also used to accelerate the engineering design cycle for robotic systems and reduce its costs. Serban et al. [57] propose Chrono, a multi-physics simulation package aimed at modelling, simulation, and visualisation of the mechanical parts of ground vehicles. Zhao et al. [69] introduce a simulation-based system for optimizing the physical structure and controllers of robots. The goal of the system is to take a set of user-specified primitive components and generate an optimal robot structure and controller for traversing a given terrain.

Lastly, simulations are used to generate at low cost large amounts of training data for the machine learning components of robots. Tobin et al. [63] trains models for object localisation on simulated images that transfer to real images by randomising rendering in the simulator. Similarly, Chebotar et al. [14] enable policy transfer to new real-world scenarios by training on a distribution of simulated scenarios. Finally, Andrychowicz et al. [4] teach a robotic arm using reinforcement learning dexterous in-hand manipulation policies that can perform vision-based object reorientation in a simulated environment.

2.2 MDE for Simulation-Based Testing

Developing model-driven solutions for the robotics domain is an established area, which has produced several results over the years [15, 16, 55]. The majority of the proposed domain-specific modelling languages deals only with specific robot functions such as perception or control, while there are some model-driven toolchains like RobotML [22], BRICS [11], SmartSoft [54], and Robochart [41] which provide multiple modelling notations to be used together when developing a robotic system. For a detailed description of different approaches to model-driven engineering of robots, the reader is referred to [44] and [18].

Despite the available literature on the application of MDE to robotics, the engineering of MRS is still inadequately investigated. Cattivera and Casalaro [13] conducted a systematic mapping study on the application of MDE to the engineering of mobile robots and they found that out of all the studies reviewed, only 19% (i.e. 13 studies out of 69) deal with MRS. The most common formalism used for modelling multi-robot behaviour is finite state machines and statecharts (e.g. [26, 47, 59]). Other approaches include Ciccozzi et al. [15], who propose the FLYAQ family of graphical domain-specific languages to model the structure and behaviour of multi-robot aerial systems, and Pinciroli and Beltrame [51] who propose a textual DSL for specifying the behaviour of robot swarms. Instead of developing a language for specifying the behaviour of multi-robot systems, Dragule et al. [24] extend FLYAQ with a specification language, which enables engineers to specify domain-specific constraints for robotic missions in a declarative manner. Finally, very few approaches propose solutions for modelling explicitly communication, task allocation, and coordination between robots with the exception of [6].

2.3 Fuzz Testing

In its canonical form, fuzz testing was applied to binary applications to execute rare code paths or to find crashes triggered by inserting invalid inputs [30]. The popular fuzzer AFL [68] produces inputs augmented with invalid characters, flipped bits, or including the insertion of known or interesting integer inputs (such as maximum values). TaintScope [66], provides tracking of how the inputs propagate through the system execution code, so that it can, for example, track the inputs that potentially influence security-sensitive or crash-sensitive program aspects such as memory allocation. Fuzzers like SmartFuzz [42] attempt to trigger a specific vulnerability (integer bugs) or dangerous unsigned conversions. This is done by maintaining a pool of test inputs, and scoring them based on the number of basic blocks executed, while rewarding those that produce the integer conversion bugs the fuzzer is seeking.

Fuzzing has been used in the robotics domain. Within the ROSIN EU project(`https://www.rosin-project.eu`) an automatic fuzzing tool for ROS 2 C++ project [40] has been developed. The tool builds on top of the AFL fuzzer and performs fuzzing with the aim to identify implementation errors that are manifested as crashes of ROS nodes. Similarly, Delgado et al. [21] propose a fuzzer that can be used to identify implementation errors. The proposed fuzzer operates on state machines [9] and generates random values as input keys of state machines. While the two aforementioned fuzzers focus on finding implementation errors, PHYS-FUZZ [67] focuses on finding hazardous scenarios by accounting for physical attributes such as robot dimensions and estimated trajectories. Also, DiscoFuzzer [53] is a novel fuzzing methodology that exploits the continuity of the physical world to automatically explore the input space and detect malfunctions in robotic software modules that lead to crashes.

There are synergies between software modelling and fuzzing reported in the literature. Model-based fuzzing is using models of the system under test such as state machines and sequence diagrams to guide the fuzzing process. For example Schneider et al. [56] proposed a model-based behavioural fuzzing approach where UML sequence diagrams are mutated by fuzzing operators to generate test data. Similarly, SMuF [39] is a fuzzer for Internet-of-Things protocols and it generates test input that covers different paths of the state machine model of the protocol. On the other hand fuzzing can be used to assess the quality of MDE tools. For example, MoFuzz [43] is a graph grammar-based fuzzer that generates sets of models to find faults in MDE tools.

2.4 Grammar-Based Fuzzing

Grammar-based fuzzing is a technique used in order to better generate syntactically valid testing inputs, and help ensure coverage of the fuzzing search space by rejecting syntactically valid programs, increasing the coverage of the testing campaign. Grammar-based fuzzing can ensure inputs have syntactic validity, and can also be augmented with constraints to ensure the validity of semantics of structured documents such as XML [3]. In programming language fuzzing well-typed terms can automatically generated to test the compiler [46]. Bonsai fuzzing [64] generates non-trivial tests by using a grammar and an evolutionary process to progressively building up the size of test cases by an evolutionary process. Superion [65] is an AFL extension that incorporates grammar parsing into an AST, and mutation of the resulting subtrees. Nautilus [5] uses a grammar in the generation of fuzzing test cases but to extend its expressiveness over a context-free grammar, allows Turing-complete scripts as an extension.

2.5 SESAME Simulation-based Testing in Respect to Related Work

Compared to the above approaches, the SESAME simulation-based infrastructure focuses on the exploration and evaluation of MRS systems via fuzz testing, incorporating feedback as to how the high-level goals of the robotic mission are impacted by the fuzzing operations selected. Therefore, it seeks to identify high-level system design faults including issues in the the overall mission design and selected scenarios, instead of low-level implementation errors.

Very few approaches propose solutions for modelling explicitly fuzz testing, performance metrics, and the experimental process of simulation-based testing upon robots with the exception of [6]. The aforementioned languages and tools focus on the specification of the behaviour and structure of multi-robot systems, while our work in SESAME focuses on fuzz testing for robotic systems. Moreover, to the best of our knowledge the SESAME testing DSL presented in this report is the first language that can be used to specify fuzzing test cases for robotics. Finally, our approach is simulator-agnostic, since its generic, message-based architecture allows it to be easily extended to accommodate experimentation with different robotic platforms.

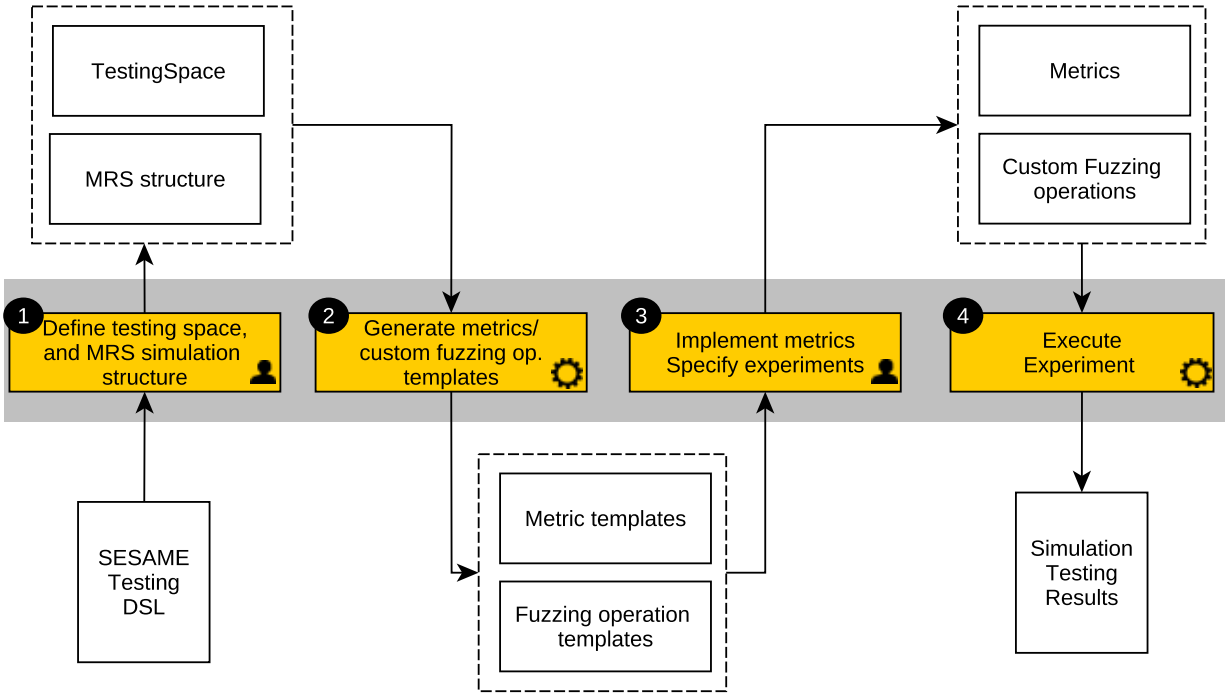


Figure 1: Simulation-based testing methodology

3 SESAME Testing Approach

3.1 Overview

The proposed approach to implement the solutions to challenges mentioned in Section 1.3 is based on a simulation testing framework that handles the whole process from the authoring phase with the transformation of the definitions of Executable Scenarios (ExSce) and EDDIs into completely configured simulation models. This is then followed by a runtime phase where simulation models are instantiated, connected to the control system, the corresponding EDDIs and to the data collectors and run to perform the acceptance tests described by the ExSce. We describe the overall simulation-based testing methodology developed as part of Task 6.2.

3.2 Simulation-Based Testing Methodology

This section describes in detail the process implemented currently for WP6 as of this deliverable, considering the application of model-driven engineering (MDE) and evolutionary optimisation for system testing. SESAME WP6 enables the evolution of effective fuzz testing campaigns that reveal violations of system safety requirements. These testing campaigns correspond to edge scenarios that can be analysed by domain experts and inform the hardening of the robotic software implementation. Identifying these edge scenarios using our currently implemented simulation-based testing framework entails following the methodology presented in Figure 1. The four steps of the methodology are labelled according to whether they involve human action or whether they are automated. The steps are presented below:

Step 1 The users specify the testing space specification and MRS structure model. We provide the specification of the testing DSL in Section 3.5. The MRS structure model corresponds to information provided by the Executable Scenarios workbench (see deliverable D3.2) which is retrieved by executing a model-to-model transformation. This MRS model encodes the characteristics of the target mission including the robotic systems, the simulation structure, and the mission requirements. Currently, in the supplied

implementation, these two models are combined into a single model resource, and the users must supply this stub scenario information. The output is a single resource model containing both the testing space and MRS system structure.

Step 2 The code generation engine employs the devised models and automatically generates mission requirement performance metric templates whose instantiation enables to assess whether a mission or safety requirement is met, and if not, the extent and impact of the violation. Since the fuzzing operations selected and requirements quantification are mission and system-specific, users are responsible for populating these templates.

Step 3 Users implement the previously defined metric templates, to create the required custom scenario-specific metrics. In particular, they specify via the testing DSL particular fuzzing test campaigns, corresponding to the particular experiments which they intend to execute. These constitute a selection of a particular set of fuzzing operations defined in the testing space, and a subset of the defined metrics used to assess requirement violations. The type of experiments, and any experiment-specific parameters are also specified. For example, when using genetic algorithms to drive the evolution process, experiment-specific parameters such as the number of generations and iterations can be included.

Step 4 The SESAME simulation-based testing platform then uses its experiment runner to perform a given experiment. The experiment runner is a system component responsible for using selected experiment details from the testing model, and dynamically generating and launching repeated iterations of tests according to a given strategy specified for the experiment. For example, we are currently focusing on an experiment runner that performs evolutionary optimisation via a genetic algorithm. The experiment runner evolves a population of tests (see Figure 2). Each test comprises a set of fuzzing operations, with each operation specifying a set of participating simulator variables, the simulation messages to be fuzzed and their characteristics (including the timing constraints and parameters).

The experiment runner which incorporates the evolutionary algorithm evaluates each test by first dynamically generating a specialised test runner which acts as a middleware, communicating with the low-level simulator via a simulation specific interface, and using any custom supplied metric definitions provided in Step 2 to quantify the impact of the fuzzing test. This information is communicated to the experiment runner and used to guide a multi-objective optimisation process. This process uses genetic operations such as mutation and crossover to create new tests, discarding the worst performing campaigns from the population. This iterative process continues until an experiment-specific termination criterion is satisfied, i.e., either the maximum number of permitted generations is reached, or no improvement occurs over a specified number of evolution rounds. Once the evolution terminates, the testing platform produces an approximate Pareto optimal set of fuzzing campaigns, along with the associated approximate Pareto front of mission requirement values. Our approach also logs all intermediate results using the framework to the model, or to files, depending on the selection in the model instance. Analysing the time series of logs, both from our platform and from the simulator itself is planned for future work as we anticipate this will reveal additional insights into the simulation and the violations discovered.

3.3 Simulation-Based Testing Architecture

The high-level architecture of the simulation-based testing framework is depicted in Figure 3. A primary component of the SESAME simulation-based testing framework is a domain-specific language (DSL), detailed in Section 3.5, that supports the specification of the fuzzing space, providing the structure of the available fuzzing operations to be performed. Employing DSLs tailored to a specific domain is a highly desirable ingredient of modern software and systems engineering practice as raising the level of abstraction enables non-experts to use the system with modest effort [13, 12, 11, 28].

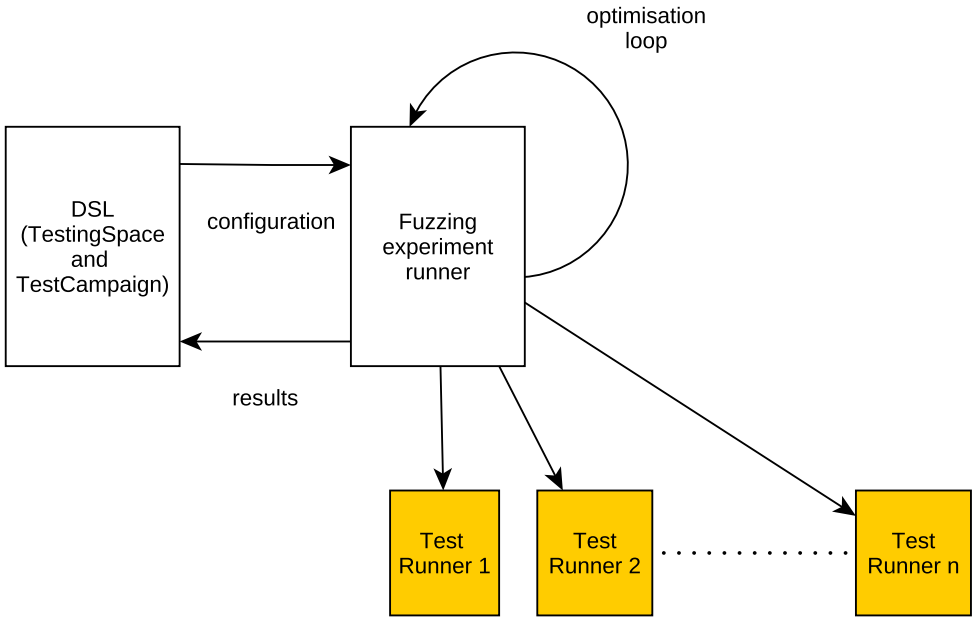


Figure 2: Code generation process, showing evolutionary experiment runner and test runner

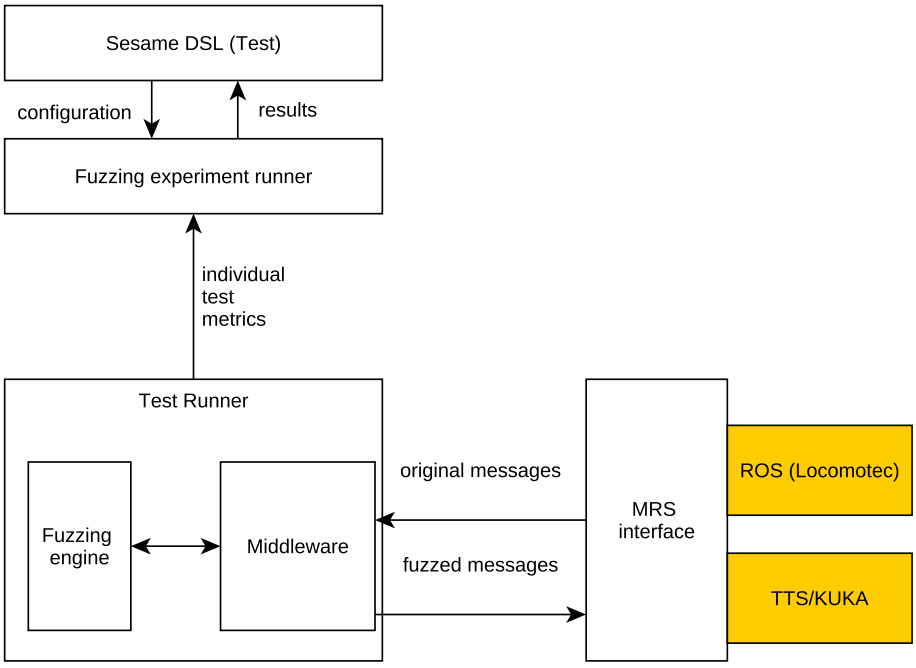


Figure 3: The architecture of the test runner and its runtime connection to the MRS

The DSL is paired with a model-driven code generation engine that consumes DSL-compliant models and generates a lightweight middleware component and a logical interface that facilitates the communication with the target robotic simulator. Furthermore, a fuzzing specification is generated that encodes the fuzzing configuration space, and which is later used by the fuzzing experiment runner to evolve testing campaigns, as detailed in Section 3.9.

The middleware fulfils a twofold role. First, it acts as the mediator between the fuzzing engine and the MRS simulator by propagating the changes instrumented by the fuzzing engine during the execution of a selected

fuzzing test. Second, the middleware enables the monitoring and recording of communication of specific message streams between components of a robotic simulator, thus supporting the quantification of properties of interest and the generation of useful insights regarding the capacity of the robot to satisfy the defined dependability requirements. To achieve these roles, the middleware leverages the modular structure of robotic systems and the publish-subscribe architecture of modern robotic frameworks (e.g., ROS [52], MOOS-IvP [8]). Through this architecture, the communication between software components and the robot interaction with the environment (by controlling its hardware resources) is driven by message exchanges using input and output interfaces [19].

On the robotic simulator side, the simulator-specific interface mediates the communication between the middleware and the target robotic simulator. Hence, this interface reduces the coupling between components and enhances extensibility. As demonstrated in recent research [33, 12], this approach provides a reusable extension point whose specialisation enables to connect and interchange different robotic simulators easily for experimentation and analysis. These features help to alleviate some of the key challenges identified for developing, debugging and maintaining robotic applications (Section 1.2).

At runtime, the architecture of test runners capitalises on this flexible and decoupled interaction mechanism with the robotic simulator to intercept messages passing internally between robot components. Figure 3 shows the communications occurring with an individual test runner and the MRS for specific test cases, and the storage of the results back into the model (via the experiment runner).

The message communication mechanism employed by the testing framework thus resembles a 'man-in-the-middle' communication setup where messages from publishing components are redirected via the middleware before reaching their subscribing components. The middleware may act as a null relay, transmitting the messages directly back to recipients unmodified, or the fuzzing engine may execute some of the available fuzzing operations, based on the configuration of the current test runner. SESAME provides several common fuzzing operations including value modifications (altering message content according to the operation definition and its parameters), timing modifications (e.g., delaying a message), or partial or complete dropping of messages (following a probability distribution). The testing platform enables augmenting the repertoire of fuzzing operations with additional custom operations tailored to the particular MRS scenario under study.

It is possible to deploy the specified architecture in different ways. The middleware can potentially run upon the robot itself, assuming it provides sufficient hardware resources and a compatible software environment to support its components. Otherwise, if the system architecture is such that there is a central publish-subscribe database running upon a fixed computer which MRS vehicles communicate with, then the middleware can be installed upon this system. In both cases, there will be a timing delay introduced, the impact of which is dependent upon the frequency of communications. In our past experimental tests of a similar architecture ([61]) we did not notice any vehicle control or stability problems resulting from this man-in-the-middle forwarding. It is possible however that some high-frequency fuzzing operations could be impacted by this, so it is important to run a baseline test to ensure the monitoring process itself does not introduce undue behavioural changes.

3.4 Fuzz Testing Dimensions and Test Coverage

There are a large number of choices in specifying a fuzz testing experiment for MRS systems. These particular dimensions which control the impact of fuzzing into an MRS include:

Fuzzing operations selected: This controls the choice of fuzzing operation (the specific method of message manipulation); for example, message content modifications, probabilistic message deletion message deletion (grayhole) and constant message deletion (blackhole). These fuzzing operations are discussed further in Section 3.6.

Fuzzing operation parameters: A fuzzing operation may have specific parameters which control its intensity, or the impact upon the MRS. For example, a fuzzing operation which involves modification of sensor detection angle may have parameters which limit the maximum intensity of the angle alterations applied.

Fuzzing activation strategies: A fuzzing operation may not be constantly active, but may be activated or deactivated at different points during the simulation. The first and simplest activation method is a fixed time range of activation, during which the fuzzing operation will be constantly active between fixed start and end times. A potentially more interesting approach is to activate and deactivate fuzzing in response to particular simulation-specific conditions. For example, fuzzing could be activated when a worker approaches within a certain distance of an industrial machine. This condition-based fuzzing approach would help to provide reproducibility, because variations in timing, either due to mission variations or instability in simulation startup time, could be compensated for. A condition-based approach would compensate for these timing variations, and could provide a more deterministic impact of MRS fuzzing over identical executions of the same system.

3.4.1 Fuzz Testing Coverage

As fuzz testing is scaled up to larger numbers of simulator variables and available operation parameter set choices, the size of the testing search space will become ever more significant. The size of the fuzzing configuration space scales exponentially with the number of potential fuzzing operations included in test. If for a potential fuzzing variable there are O operations with P viable parameter choices, then the fuzzing operation has OP possible configurations. If V variables upon distinct robots each use this fuzzing operation set, then there are now $(OP)^V$ potential configurations for the fuzzing search space.

However, the total size of the fuzz testing space is actually greater than this, because this assumes fuzzing is constantly active. If time-based fuzzing activation is used, then there will be a large number of T temporal choices for each fuzzing operation and variable (assuming the temporal dimension of fuzzing choices is discretized). If any fuzzing choices are permitted to occur at any timing point with no dependencies between them, then the search space consists of $(OPT)^V$ test configurations.

The coverage of the testing space can be quantified. If parallel evaluation of test configurations is permitted upon M machines ($M = 1$ is a possible configuration, corresponding to serial evaluation on a single machine), and each evaluation takes S seconds, then the coverage C at time t can be quantified as in Equation 1:

$$C(t) \approx \frac{M \times t}{S \times (OPT)^V} \tag{1}$$

Large values of O, P and T, and especially V for a particular experiment constitute a very large search space, and when performing simulation-based testing, the long execution time of simulations (potentially corresponding 1:1 with wall clock time) renders exhaustive coverage of the space prohibitive.

It is therefore important to use the expertise of system testers to constrain the space, perhaps by testing early cases with a restricted number of robots selected by these engineers as most likely to produce violations (reducing V). Our system testing methodology allows this reduction by specifying the boundaries of viable fuzzing operations, activation timing and parameters ranges for the entire space. The model also allows the user to specify a subset of these for a particular test campaign, which allows more focused experiments. Further details upon the metamodel and how it is applied to system testing is presented in Section 3.5.

Despite the reduction of the search space through the specification of a specific testing space and campaign, it is necessary to search intelligently and exploit the fact that similar configurations often have similar behaviour. An evolutionary approach is further used to concentrate the search on the most interesting regions of the space, iteratively modifying the previous tests in response to multi-objective feedback from the simulation. This is detailed in Section 3.9.

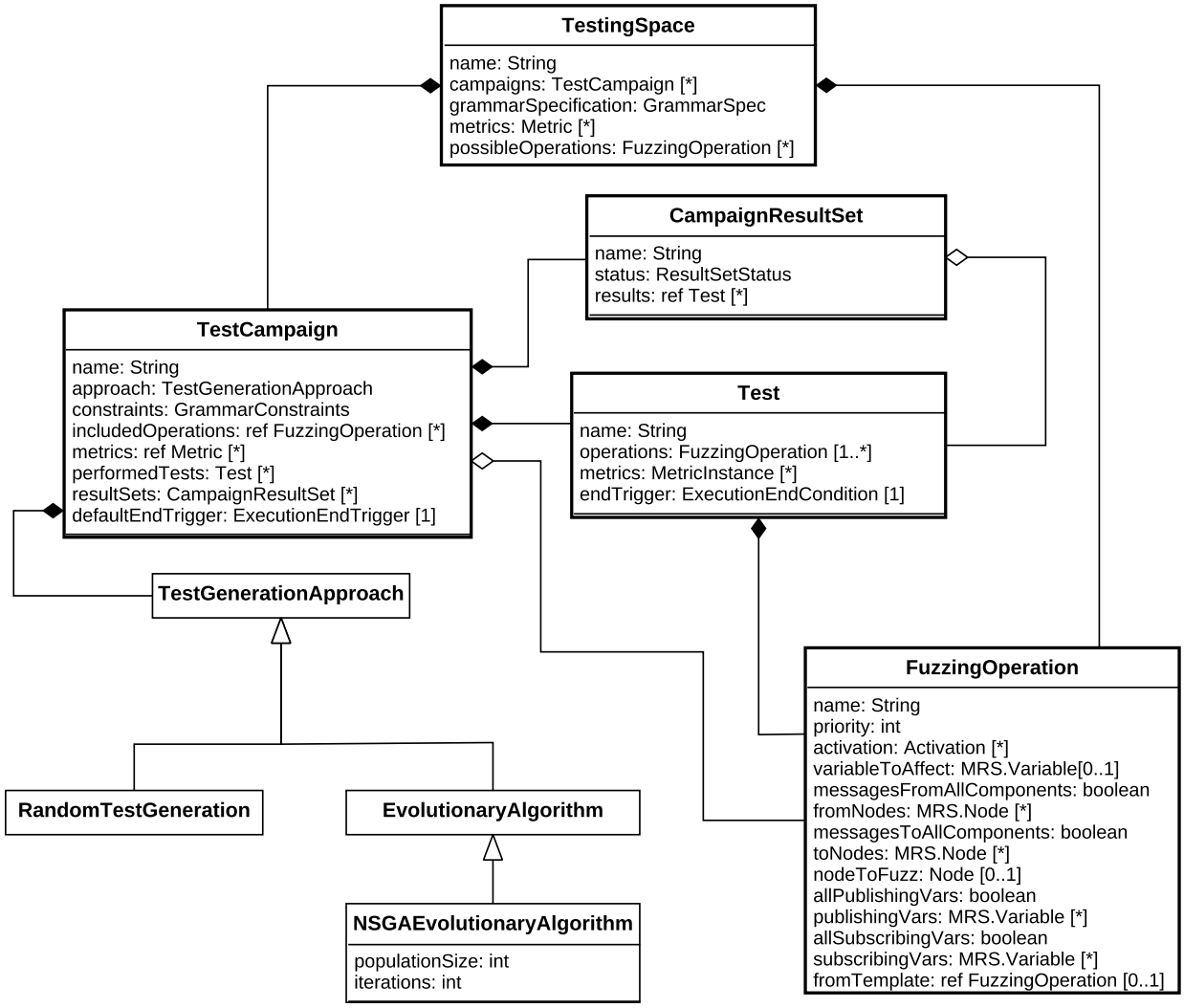


Figure 4: UML classes for the core elements of the testing space DSL

3.5 Testing Specification Domain-Specific Language

A Domain-Specific Language (DSL) is used to specify the structure of the fuzz testing system, the associated fuzzing campaigns, and the testing space. The usage of a DSL is a viable technique for extending the applicability of the platform, since it provides an increased level of abstraction, allowing end-users such as test engineers who are not software developers to make more effective use of our platform, by leveraging existing model-based development tools and techniques. It also provides the opportunity to interconnect other project tools with the simulation-based testing framework, by model-to-model (M2M) transformations. In addition, our testing platform provides a convenient visual editor for system test engineers to configure the MRS system testing experiments.

A UML diagram showing a fragment of the DSL with three of the main classes included is presented in Figure 4. The figure presents the TestingSpace, TestCampaign and Test classes. TestingSpace is the root element of the DSL, and consists of a set of permissible fuzzing operations (possibleOperations) that collectively specify the boundaries of the potential fuzzing space that can be explored. The fuzzing operations that can be applied to the robot system in a particular fuzzing test are always a subset of these, with potentially more specific parameters. The testing space also includes particular metrics which are used to quantify the robotic system performance in regards to safety violations, which allows multi-objective optimisation of system performance

to be performed. References to a grammar are reserved to specify a grammar for custom fuzzing conditions, that control the activation and deactivation of particular fuzzing operations.

The TestCampaign class specifies an experiment that can be performed and sets parameters for a specific fuzzing experiment. A TestCampaign references a choice of particular metrics to use in evaluating that campaign. The includedOperations reference list allows the selection of particular operations in order to constrain an experiment, by allowing a system test engineer to choose interesting or relevant operations. The TestGenerationApproach allows the user to specify the parameters for an experiment by selecting one of these subclasses. For example, including NSGAEvolutionaryAlgorithm allows an evolutionary experiment with the NSGA-II algorithm [20], and contains specific parameters relevant to this the experiment, e.g., the number of iterations and the population size. The performedTests attribute is populated during the execution of experiments, containing the particular Tests generated and executed for that campaign. The resultSets attribute is also populated as the experiments proceed.

The Test class represents one test configuration that can be applied to the MRS, corresponding to a particular selection of operations, and the recorded history of evaluation of performance metrics. The latter are represented by the containment of MetricInstances, which record performance metrics for the results of evaluation of that particular Test. During execution of the Test, the metric instances will be recorded and stored within the model. This provides a record of the impact of the fuzzing performed. Further details of the performance metrics will be presented in Section 3.7.

The FuzzingOperation class represents one specific fuzzing operation - an entity which represents a specific strategy for making runtime modifications or disruptions to the MRS. Subclasses of FuzzingOperation are used to represent the specific semantics of this fuzzing operation, for example, the PacketLossNetworkOperation class represents probabilistic loss of a proportion of packets. Currently, *variableToAffect* is the main attribute used when setting up variable subscriptions and defining the variables to which fuzzing operations are applied. Section 3.6 discusses these subclasses in further detail.

The CampaignResultSet class represents either the final outcome of a particular campaign, or partial intermediate results obtained during its execution. An enumeration, ResultSetStatus, which is set to either FINAL or INTERMEDIATE, determines this status of a result set. Result sets contain references to particular Tests that comprise the results. For example, in an evolutionary experiment, the set of Tests for a final result set would contain the Pareto front obtained during an experiment. Intermediate result sets would allow the user to investigate the change in the front at particular generations as the experiment progresses.

3.6 Fuzzing Operations Specified within the DSL

This sections describes particular fuzzing operations that can be specified via the testing DSL. Each fuzzing operation involves a manipulation of the simulator internal state, system communications, or the resources available for simulation execution. The following categories of fuzzing operations are available:

State Fuzzing Operations: involve fuzzing disruptions impacting the simulation process itself. Examples of state fuzzing operations include for example the killing of a system process required for the simulator to function, simulator time disruptions (time skipping by incrementing simulated time, or causality problems by stepping back in time) and the failure of a node involved in the simulation.

Network Fuzzing Operations: involve the disruption of data transmissions corresponding to specific simulator messages interchanged between MRS components. For example, message disruption can involve the guaranteed or probabilistic removal of a message transmitted from one modelled MRS entity to another.

Resource Fuzzing Operations: operations affecting the ability of the host system in order to execute the simulation itself. For example, a resource fuzzing operation can involve the addition of CPU load to the simulation, in order to see how the simulation process responds to this fuzzing operation.

Fuzz Testing Operations: This category of operation involves the modification of replacement of structured parameters within a message with newly generated random values.

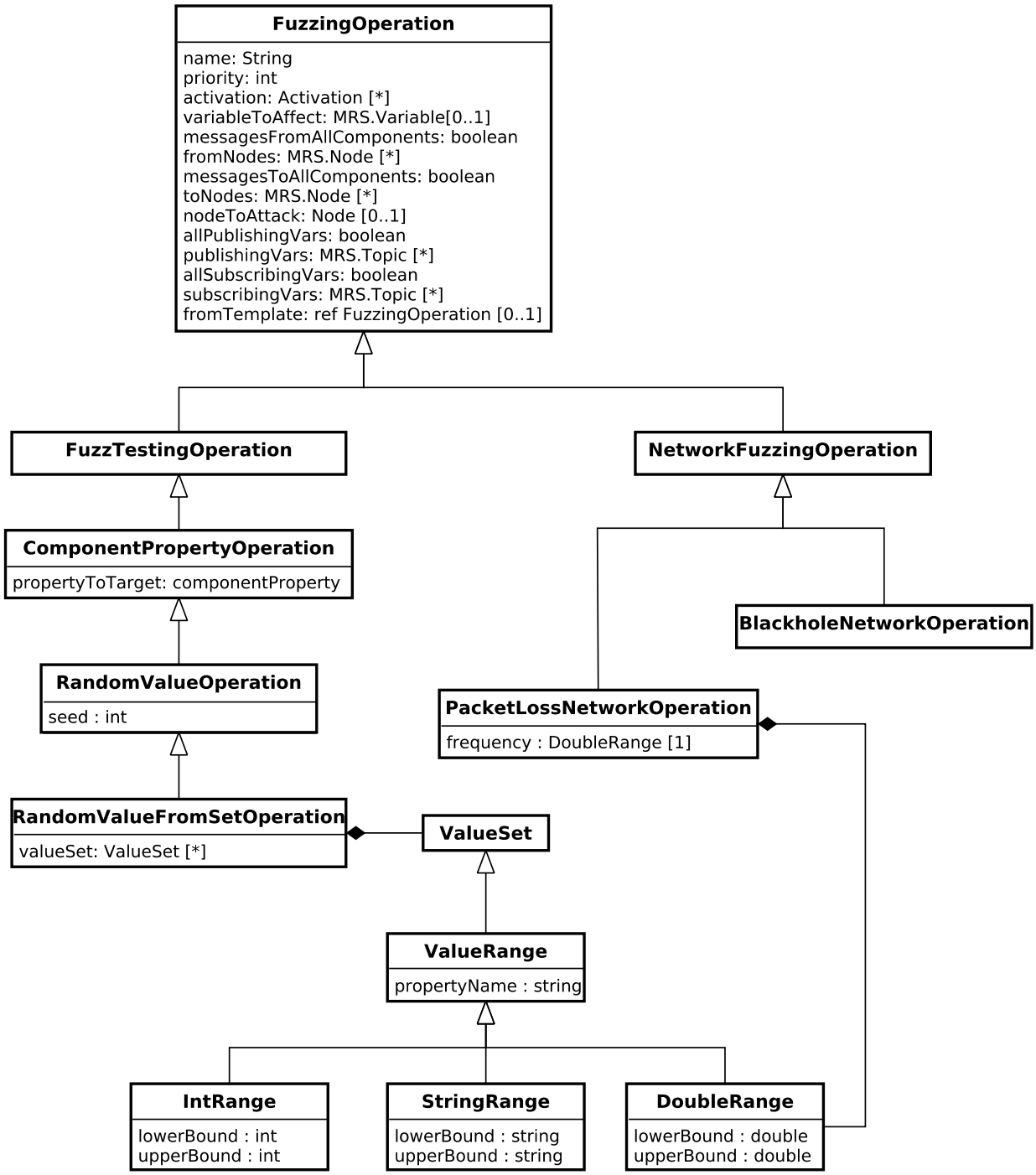


Figure 5: UML class hierarchy for three implemented fuzzing operations in the DSL

Some specific categories of fuzzing operation are presented in the class hierarchy in Figure 5. These operations all extend from the base class FuzzingOperation, with additional properties:

BlackholeNetworkOperation: This class defines a network fuzzing operation which involves the disruption of message transmissions from one node to another. When a blackhole fuzzing operation is active, all transmissions on a particular simulator variable will be dropped by the fuzzing engine, preventing the message from reaching its intended destination(s). This fuzzing operation may correspond to realistic simulation scenarios in which for example, a mistuned transmitter, significant interference, or cabling with intermittent fault causes the breakage of a communications link.

PacketLossNetworkOperation (greyhole): This is a network fuzzing operation which involves the disruption of message transmissions in a similar manner to the blackhole fuzzing operation. However, the primary difference between them is that the grayhole fuzzing operation is probabilistic and performs a random test before discarding the message. A configurable parameter is provided in the DSL which allows the range of loss probabilities to be specified in the testing space. A specific implementation of this, in a Test in a TestingCampaign will have the lowerBound equal to the upperBound. This value will be tested independently for every message passing through the testing infrastructure in order to determine the probability of the message being lost.

RandomValueFromSetOperation: This fuzz testing operation permits the replacement of structured parameters within a message with newly generated random values. For example, an MRS vehicle command velocity can be replaced with randomised values, modelling a situation in which a malfunctioning component or corrupted message leads to an incorrect position data, in order to study the resulting effect upon position. Alternatively, for a sensor used to detect humans in a topology, randomised data can be introduced to the human orientation angle in order to model a scenario in which an individual's location is incorrectly detected.

The testing DSL permits a mechanism for the RandomValueFromSet fuzzing operation to specify generic parameters for randomisation, allowing it to be flexibly configured for novel variables and custom scenarios. A RandomValueFromSetOperation can contain multiple ValueSets, and each value set specifies the upper and lower valid ranges of the parameter. In turn, the propertyName attribute specifies which message component is altered by randomised fuzzing at runtime.

Additional classes, supporting additional functionality such as message duplication or variable delay leading to messages arriving out-of-sequence will be considered in subsequent releases of the simulation-based testing platform.

3.7 Performance Metrics

When performing simulation-based testing incorporating fuzz testing, it is important to quantify the impact of fuzzing tests upon the system. This impact assessment can be both functional (affecting the mission requirements), and non-functional (impacting safety requirements which the system must observe in order to avoid causing hazards to either humans in the vicinity or to other systems). Fuzz testing operations can cause both partial or complete failure to complete the mission, and violations of safety requirements. For example, a fuzzing operation may cause a safety violation by increasing the power applied to a light source, causing a hazard to a human in the environment. It may also cause mission failure by causing a robot to overspeed, which increases energy consumption and causes batteries to expire before completing the mission. Performance metrics permit the quantification of these violations, for example, defining the number of occasions in which a human was exposed to excessive radiation during a UVC disinfection procedure, or the total dosage delivered. Alternatively, they may define the number of objects detected during an inspection mission. This would allow trade-offs between mission completion and safety violations to be assessed.

Two types of metrics are supported in simulation-based testing. In some cases it is possible to obtain metrics directly from the MRS simulator state, for example, because a simulation component already directly computes the metric as part of its own state. In this case, the metric is referred to as a variable-based metric, and the DSL definition names the variable to be monitored to obtain the metric value. One example would be in the case of an aircraft inspection mission, in which nodes must compute their distance from the aircraft surface and count incidents of excessively close approach. The simulation can be augmented by test engineers adding a new component (e.g., in the case of a ROS simulation, a ROS node) in order to monitor the current 3D distance and return the metric as a count of close approaches. It would be convenient to do this given that the simulator has the necessary state for computing the 3D distance of the vehicle to the aircraft surface. With variable-based metrics, the monitoring tool does not require any additional processing to compute the metric, since it merely reports their values directly.

Another approach is to compute performance metrics within the testing infrastructure itself. These are referred to as stream-based metrics, because they are computed via processing the stream-based infrastructure which

handles the data obtained from the simulator. These metrics can be computed using custom code supplied to the infrastructure by the system testers. An example of these would be to measure the actual message loss ratio of specific messages, by comparing the number of events on the middleware INPUT and OUTPUT streams, and thus determining the actual statistics of packet loss ratios resulting from fuzzing applied. These necessarily incorporate information from multiple simulator variables, each of which corresponds to different events. In the case of a stream-based metric, the system test engineer is expected to supply the implementation for the class which will compute the metric.

In both cases, metrics can specify a set of simulator variables that must be monitored at runtime for the computation of each metric to which the test runner must subscribe. For a variable-based metric, this set of variables is merely a single element; the subscribed variable itself. For stream-based metrics, multiple variables can be selected within the DSL, in order to ensure that the metric will receive the necessary events to use all relevant state for their computations.

Both variable-based and stream-based metrics are transient and change over time during the evolution of the simulation. The testing platform supports logging these metrics to a file, showing their full history over the development of the simulation.

3.8 SESAME Test Interfacing

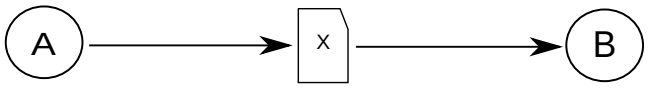
In order to evaluate each individual test, the SESAME simulation-based testing platform generates custom test runners for each individual test using MDE, which incorporates a middleware for connectivity to the MRS at runtime. The MRS is invoked by this test launching code, and communicates with the test runner via a simulator-specific interface. The experiment runner also listens to performance metrics provided from the test runner (as described in Section 3.7), and stores the metrics within the relevant MRS test model, under each test. This provides an auditable history of the outcome of particular tests.

When a test runner aims to manipulate the MRS internal state, it uses a man-in-the-middle approach to modify the message flow between components, in which the message flow between source and destination MRS components is intercepted. The test runner/middleware dynamically modifies these messages at runtime, potentially discarding, delaying or modifying them according to the configuration of the particular test.

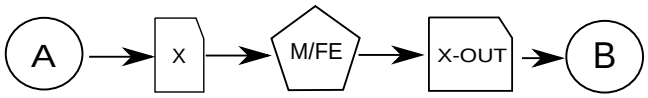
Consider a situation in which components A and B of the MRS communicate via a variable X. This is illustrated in Figure 6a. In order to perform simulation-based testing, this topology must be modified. There are two approaches provided for simulation interfacing, named for the directions in which the simulation must be reconfigured to support the communications. The first is referred to as OUTBOUND, because the reconfigured component must receive information on the simulator output. For example, when a subscription is made to variables published by simulator component A, the simulation configuration must be modified to ensure that the original destination components B only listens to alternative variables with OUT appended to their names. The middleware will listen to the original variable X, transform the messages according to the given fuzzing operation(s), and republish the modified values upon X-OUT. This is illustrated in Figure 6b.

An alternative is provided in situations in which the destination components cannot be reconfigured, and instead it is intended to reconfigure the sending component, so the testing platform will receive input from the reconfigured component. In this case, the sending MRS component will be reconfigured to transmit upon X-IN, and the middleware will receive this and transmit the fuzzed messages upon the variable name X. This supports scenarios in which it is easier to reconfigure the MRS sending component. This is illustrated in Figure 6c.

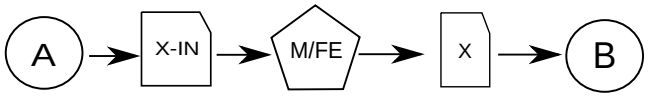
In simulations that are sufficiently flexible, it may be possible to add a standardised interfacing component, and modify the simulation configuration in order to implement these message-based fuzzing operations. For example in ROS, adding a *rosbridge node*, and manipulation of the ROS topic graph, either dynamically or via modification of static launch files, can be sufficient to provide this reconfiguration for simulation-based testing. In other simulator frameworks, it may be required to implement custom code in order to provide this



(a) Normal flow of information between components A and B via variable X



(b) Interfacing via the middleware for simulation-based testing (OUTBOUND)



(c) Interfacing via the middleware for simulation-based testing (INBOUND)

Figure 6: Simulation interfacing for remapping. Circles illustrate the simulator components, the cut rectanges illustrate the simulator variables, and the pentagon the added simulation-based testing infrastructure

interface for message manipulation, or manipulate simulation internal source code to reconfigure the system to support the necessary remapping. Some fuzzing operations will require an additional MRS module interfacing with simulation internals if the simulator does not provide a standardised interface; especially, fuzzing operations that involve manipulating system time in non-standard ways, e.g., potentially violating causality would necessarily require platform-specific simulator support. There may be a requirement for simulator source code modifications to implement this, if the MRS simulation does not provide sufficiently flexible APIs to support making these reconfigurations.

3.9 Fuzzing Engine and Fuzzing Evolution

There are a variety of strategies possible for triggering fuzzing. The currently supported strategy by the simulation-based testing platform uses the MRS system clock to determine when fuzzing is active. This activation strategy is known as “time-based fuzzing” and is the currently supported approach to fuzzing within the simulation-based testing framework. As described in Section 3.8, executing simulation-based testing campaigns is underpinned by the realisation of a ‘man-in-the-middle’ orchestration principle. If fuzzing is activated on a given variable (i.e., the MRS system clock specifies that fuzzing is activated then), then the selected fuzzing operation is applied and the modified variable is propagated to its recipients; otherwise, the original message is transmitted unmodified. Adopting this strategy enables applying fuzzing while minimising changes to the internals of software components. A key question for uncovering violations of safety requirements for a MRS system is how to determine these time intervals during which the system is vulnerable to fuzzing, especially given that exhaustive coverage of the testing space is intractable.

Using as input the fuzzing specification model that defines the fuzzing space (and conforms to the metamodel from Figure 4), our time-based fuzzing approach evolves a population of fuzzing tests aiming at establishing the critical moments (in the temporal dimension) where safety requirements violations occur. To achieve this, time-based fuzzing activation uses the simulator’s timing reference and statically defined start and end times, as specified in the evolved fuzzing test. When the start time for a variable within the fuzzing campaign is reached, the fuzzing operation becomes active and is applied to messages upon that variable. Once the end time is reached, the fuzzing operation becomes inactive, and messages will be forwarded to their recipient components unchanged.

The fuzzing engine employs a pipeline which allows multiple interleaved fuzzing operations to be active simultaneously, even upon the same variable, with potentially overlapping time ranges and compounding effects.

When multiple fuzzing operations are active simultaneously, their effects are applied in the order specified in the fuzzing test. This is because fuzzing is implemented internally as a stream transformation, enabling to produce complex fuzzing effects whose overall effect is the amalgamation of the individual fuzzing operations selected.

Our simulation-based testing platform supports multi-objective genetic algorithms [17] for test campaigns with automatic generation of informative fuzzing tests. The precise choice of evolutionary algorithm selected for an experiment, and its parameters is configurable within the *TestingCampaign* within the resting DSL (Section 3.5). We currently support the NSGA-II genetic algorithm [20]. To use it, the user should select NSGAEvolutionaryAlgorithm as the test generation approach for the selected TestCampaign.

The evolutionary algorithm manages a population of tests, with each test consisting of a set of fuzzing operations. Each operation is constrained by the testing space in regards to each of their dimensions, such as timing activation, and specific intensity parameters and the metrics chosen. For example, if the testing space indicates the probability of a fuzzing operation between 0.1 and 0.5, then any element in the population can consist of individual fuzzing operations specialised by the algorithm selecting a probability within this range. Each operation has activation conditions, corresponding to its time range of activation, and is specialised to a particular variable (e.g., disrupting one message communication on one robot).

Generally, genetic algorithms combine the initial randomised generation of the tests with incremental improvement, aiming to increase the intensity of violations found, as measured by the output performance metrics. Firstly, the best tests (in terms of violations produced) are selected from the initial population and combined, merging fuzzing operations from the parent tests in order to create children. This aims to combine features from the best performing tests that may contribute to the test's effectiveness. These children are then mutated by alteration of the fuzzing operations within the tests, in order to add additional random variation to the child elements. The newly generated children are evaluated, and the worst performing elements are evicted from the population. This process is repeated until completion of the algorithm, either by a fixed number of generations or some performance criterion related to improvement, until the application terminates and the output Pareto front of best configurations is returned.

Given the fuzzing testing space and selected testing campaign information in the model, the simulation-based testing framework produces an initial population of fuzzing tests. Initialising each new individual entails iterating over all fuzzing operations available in the fuzzing specification model for that campaign, and selecting a subset of operations to included in this campaign. During reproduction, specific crossover and mutation operators are employed to create a potentially improved population (offspring). In particular, crossover is applied between any two individuals from the population by swapping fuzzing operations through one-point crossover [17].

For each generated offspring, mutations are applied to each of the potential fuzzing dimensions of fuzzing operation parameter, and timing based on each dimension's independent mutation probability. Mutating operation parameters involves generating new parameters for the operation, as specified by the DSL testing space. For example, mutation of operation parameters for a velocity fuzzing operation will change the velocity vector, and mutation of a time delay operation will change the time delay applied. Mutation of participants consists of selecting a new random set of robot participants. Similarly, mutation of fuzzing timing involves replacing the start and end times with new randomly generated values while also respecting the timing constraints for these fuzzing operations specified in the fuzzing testing space. This evolutionary process favours the configurations that produce extreme values of particular metrics (either highest or lowest, depending on the interpretation of the metric, whichever indicates the highest intensity of violations). Once the evolution terminates, the Pareto-optimal set approximation corresponding to the best found fuzzing campaigns and metric values, is stored in the DSL.

One important factor to consider is the criterion for ending the search of the fuzzing space. The currently implemented option is to run the evolutionary search for a fixed number of iterations, which is the simplest possible termination criterion. In future work, we aim to extend this by using a more advanced termination condition. One approach is to monitor the in-progress Pareto front, and to terminate the experiment when the

rate of improvement is either zero or below a certain threshold. Another alternative is to use the coverage metric specified earlier to ensure that the search has covered a given proportion of the available testing space. For example, searching until a given fraction of the space has been covered may be an option for long-running experiments perhaps performed on a cluster or server.

3.9.1 Reproducibility

One of the key factors leading to non-determinism in the simulation is related to timing offsets. In some of our previous research, we were able to compensate for the simulation start-up time variation in real experiments by using a custom time reference, which would only activate at time zero when the mission was ready to begin [61]. Therefore, supporting a generic simulator interface, which allows a custom time reference for a particular simulation can allow such timing stabilisation to be achieved. This generic simulation interface is defined in Section 4.1.

Another approach which can be performed is to analyse the determinism of experiments, by using an experiment runner which supports multiple executions of the same configuration, and to use search algorithms focused upon isolating the rare or unstable events. A further approach to improving determinism is to trigger fuzzing events in simulation-based testing based upon timing-independent conditions, upon the detection of events that are major contributors to deviations in safety requirements. For example, a fuzzing operation can be activated when the distance of approach of a human to a robot is less than a fixed threshold. This would reduce the dependency of our fuzzing results on precise timing by the simulator and operating system. It will also increase the applicability of the simulation-based testing platform to missions supporting non-deterministic actors, that exhibit significant variability in behaviour between multiple executions of the same fuzzing test configuration. We plan to implement this condition-based fuzzing approach as the next stage in development of the simulation-based testing platform. A description of this condition-based fuzzing technique will now be provided.

Condition-based fuzzing in the context of simulation-based testing aims at discovering critical events leading to deviations in safety requirements. A condition corresponds to a state of the robotic system during mission execution that incorporates semantic information upon the mission and system components, and supports the specification of complex spatial and geometric properties. Evaluating conditions at runtime enables to determine whether it holds or not. For instance, in the Locomotec disinfection use case [48], a condition for activating fuzzing could be “the robot’s distance to a human is less than 1 metre”. Its evaluation requires using simulator variables such as the robot and human positions in the topology to establish if the condition is currently true or false.

The premise underpinning condition-based fuzzing is the evolution of semantically meaningful testing campaigns. Conditions are used to trigger the activation and deactivation of fuzzing, replacing the starting and end times of time-based fuzzing activation. Instead of the arbitrary metric of time, conditions are based on expressions using simulation-specific variables. Our hypothesis is that conditions are more informative and require less introspection in order to understand the overall effect of the fuzzing campaign, by stating directly what relationships between variables can result in a safety violation. Furthermore, conditions restrict the search space, making the evolution of testing campaigns more tractable (compared to the use of time which is a continuous variable).

With our planned approach to this condition-based fuzzing, a condition may have a basic format, denoting a comparison of a variable to an expression, or a composite format, consisting of multiple conditions connected with logical operators (AND, OR). An expression can be a variable or a terminal symbol (e.g., an integer for simplicity). Condition-based fuzzing will use the testing DSL to specify a set of grammar variables that are available to use in evolving fuzzing conditions. Since these grammar variables are mission-specific, their definition requires input from domain experts. At the same time, however, these variables (and their maximum and minimum value ranges) will provide the only mission-specific part of the grammar. This important character-

istic enhances the generality of the approach, enabling its application to other domains and missions without altering its core production rules.

In a currently under-review research paper [32], we have applied the technique of condition-based fuzzing to an example scenario of UAV inspection. Condition-based fuzzing was discovered to be able to identify specific points in the fuzzing space which were vulnerable (for example, upon the start of the fuzzing mission when the separation between UAVs increased between a certain threshold) as well as reducing the variance of some output metrics in a scenario in which a collision with the vehicle occurred. This improvement in determinism resulted in the fuzzing occurring at a more precisely controlled event, when the UAV approached a specific distance to the nose of the aircraft surface under inspection. We expect that the condition-based fuzzing will provide corresponding advantages for the SESAME use cases [48].

3.10 Integration with EDDIs

EDDIs consist of security or dependability-relevant models, which are transformed into executable code, capable of monitoring and responding to safety and security related events at runtime. These monitoring components can, for example, monitor the runtime condition of the system in order to detect a security attack or the system condition likely to lead to a safety violation, and dynamically trigger mitigation actions in order to avoid the condition or eliminate the hazard.

As shown in SESAME deliverables D4.3 [50] and D7.1 [49], fault tree analysis can be used to identify the vulnerable components in the system, and to quantify the likelihood and impact of failures on the system. This can feed into the simulation-based testing methodology presented in this report by helping test engineers identify the components/simulator variables that are safety- or security-critical, In this case, it may be productive to target specific fuzzing effort upon these components in order to uncover additional dynamic behaviour not anticipated in the fixed categorisation of these attacks.

Fuzzing operations can be specified and targeted in such a way as to simulate potential malicious attacks. Adding a new custom fuzzing operation with behaviour which parallels the characteristics of known attacks (but with the possibility for random variation in parameters) may reveal further requirement violations. Furthermore, the conditions identified from the fault trees in D4.3 [50] can be used in as criteria for activating fuzzing. We can concentrate on the conditions leading to the most severe hazards in order to use the simulation-based testing time most effectively.

The simulation-based testing platform can also be used to determine if the system runtime EDDI will correctly identify the violations of the conditions it is intended to detect. If a mitigation mechanism is available, our platform can be used to verify that the EDDI performs the expected mitigation for one of these fault scenarios, potentially, when fuzzing occurs in addition to the intended attack. For example, in the Locomotec use case, if fuzzing errors in human or robot localisation are applied at the time the platform is supposed to switch off its disinfection light, a false negative can occur if the system does not perform this mitigation due to transient localisation errors. This scenario could be detected by the proposed simulation-based testing approach.

4 Implementation

4.1 Methodology and Project Structure

The simulation-based testing infrastructure is implemented as a set of Java projects and tooling integrated with Eclipse, building upon open-source and widely-used model-driven engineering tools such as the Eclipse Modelling Framework[1], Epsilon[2] and Emfatic[3]. The modelling technologies selected are compatible with those used in the EDDI modelling, which will assist in integration efforts with EDDI work packages WP4 and WP5. Additional standard Java technologies such as the Maven build tool[4] are used to recompile code components dynamically generated during the execution of the experiments. Apache Kafka[5] and Flink[6] are used for message communication and stream processing, in order to interconnect the MRS simulator, the individual test runners, and the fuzzing engine that manages the experiments. Flink and Kafka were selected as they provide a standardised and mature framework for stream processing, permitting functional and stateful message transformations that are required to implement fuzzing operations and performance metrics. They are also proven scalable and can interconnect multiple processes as Kafka endpoints, supporting future extension to the parallel execution of multiple experiments simultaneously. The code developed by the project will be released as open source upon project completion, and is currently available at `https://github.com/sesame-project/simulationBasedTesting`.

The simulation-based testing implementation carried out so far is structured as several interdependent Eclipse projects. All projects (except for the generator project) are Maven based, allowing their dependencies to be automatically downloaded, and recompilation to be automatically performed. A package diagram showing the dependencies between these projects is presented in Figure 7.

- uk.ac.york.sesame.testing.architecture
- uk.ac.york.sesame.testing.architecture.ros
- uk.ac.york.sesame.testing.architecture.tts
- uk.ac.york.sesame.testing.dsl
- uk.ac.york.sesame.testing.generator
- uk.york.sesame.testing.evolutionary

In the context of MRS, a major aspect aspect of extensibility is the implementation of new simulator interfaces, allowing the simulation-based testing platform to connect to and fuzz other MRS simulators. Currently, the package `uk.ac.york.sesame.testing.architecture.ros` contains an interface to ROS/Gazebo simulations. An in-progress interface implementation for the TTS simulator used in the KUKA use case is present in package `uk.ac.york.sesame.testing.architecture.tts`. The architecture presents an interface, ISimulator, which users must implement to connect their simulator with the provided infrastructure. The purpose and structure of some of the important system packages is summarised in the following sections.

4.1.1 Project uk.ac.york.sesame.testing.architecture

This project contains the ISimulator interface which the users must implement in order to interconnect an MRS to the testing framework. This package is stored under the package `uk.ac.york.sesame.testing.architecture`. A UML package diagram of the project, showing some key classes and packages and their relationships, is presented in Figure 8.

[1] https://www.eclipse.org/modeling/emf
[2] https://www.eclipse.org/epsilon
[3] https://www.eclipse.org/emfatic
[4] https://maven.apache.org
[5] https://kafka.apache.org
[6] https://flink.apache.org

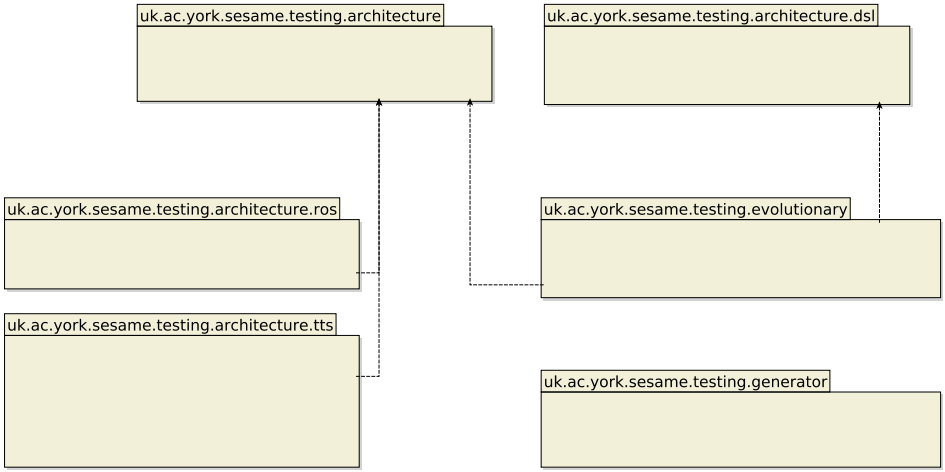


Figure 7: Package diagram showing dependencies between packages

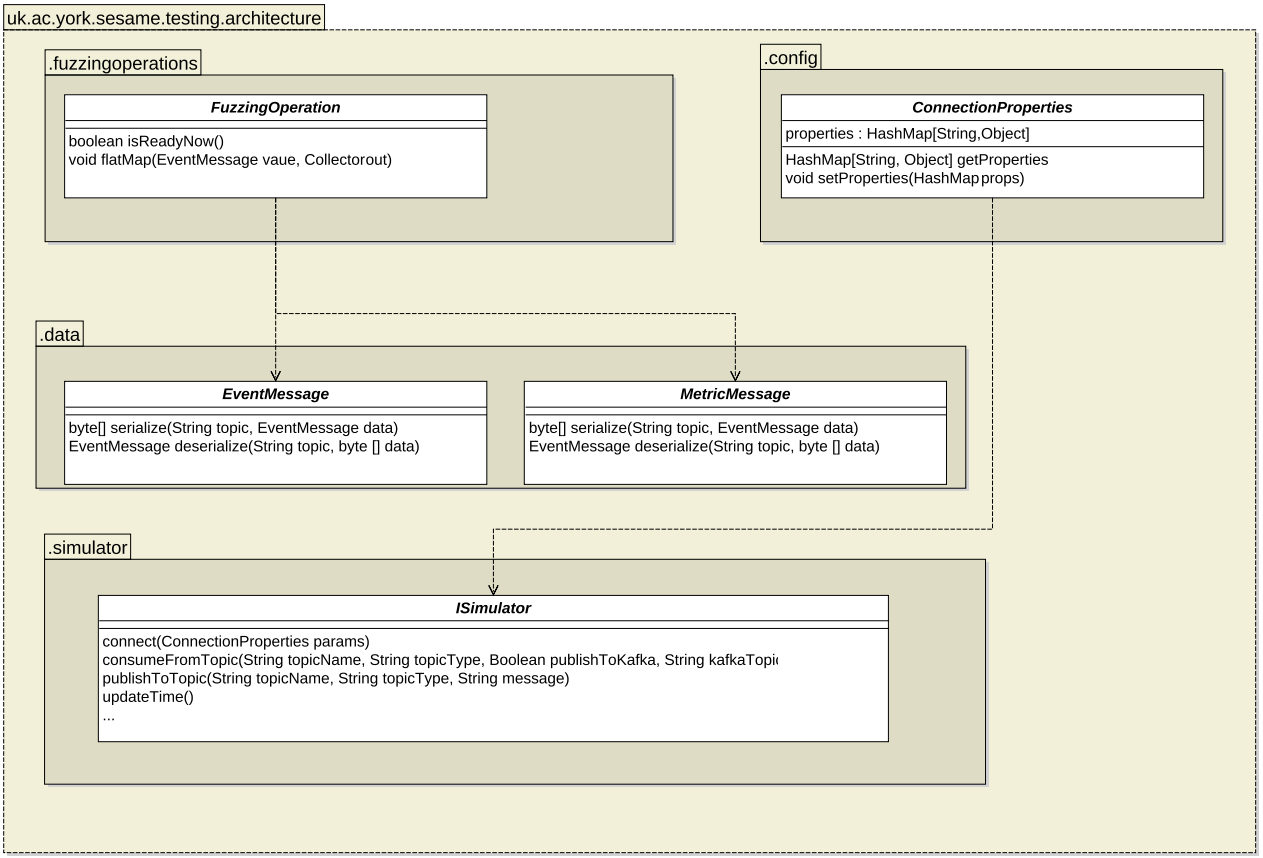


Figure 8: Package diagram showing packages and classes within the architecture project

Key methods related to MRS communications that are needed to implement this interface are presented below:

public Object connect(ConnectionProperties params) This method provides the code for connecting the middleware to the MRS simulator (e.g., for ROS, making a connection to the simulator via *rosbridge*. The parameters which are supplied are used to configure the connection, for example, supplying the hostname and port for MRS connection.

public void consumeFromVariable(String varName, String varType, Boolean publishToKafka, String kafkaTopic) The implementation of this method provides a mechanism for consuming the variables of the

robotic system by the testing platform. The first parameter is the name of the variable in the robotic system that we need to consume, the second is its type. The third is used to define if incoming messages should be redirected to Kafka (should be always true in practical use, as Kafka is used internally for message processing). The last parameter declares the name of the Kafka topic to receive the message.

public void publishToVariable(String varName, String varType, String message) The implementation of this method provides the mechanism to publish a message back to a variable of the robotic system. The parameters specify the variable name, its type and the message itself (as a String).

public void updateTime() By implementing this method, users provide the mechanism to update the timestamps in the architecture by collecting them from the underlying mechanism of the MRS (e.g., for ROS, by monitoring the `/clock` topic).

One package in this project contains some fundamental data structures; `uk.ac.york.sesame.testing.architecture.data` includes different classes EventMessage and MetricMessage which provide all the necessary code for creating, publishing and consuming messages to the Kafka infrastructure of our approach. These classes are responsible for the serialisation and deserialisation of both MRS messages and metric messages respectively.

The package `uk.ac.york.sesame.testing.architecture.fuzzingoperations` includes all the fuzzing operations that our architecture already supports, or will support. The constructor for each fuzzing operation, has as parameters all the necessary properties required for this fuzzing operation.

4.1.2 Project uk.ac.york.sesame.testing.dsl

In this project, we store the models of the DSLs involved in the project (i.e., the Testing DSL and our version of the MRS DSL). The current version of the Testing DSL described in a previous section is stored as Emfatic (under the TestingMM.emf DSL). The MRS DSL metamodel is currently integrated with the TestingMM DSL, with the MRSPackage combined. In the uk.ac.york.sesame.testing.dsl.modelGeneration package you can find the EMG file to generate random Testing DSL instances (see previous section for details on that).

4.1.3 Project uk.ac.york.sesame.testing.evolutionary

This package provides a support for evolutionary experiments using the JMetal framework, supporting population generation, mutation and crossover operators and interfacing with the testing model, adding the newly generated Tests to the model. It also stores the metrics communicated back from the model generated test runners back into the model under the relevant Tests.

The package also contains a number of scripts which work to support the experiment runner, for example, supporting automatic compilation of newly generated Maven projects containing the generated test runners, launching these test runners automatically, and terminating the simulations and test runners after execution.

4.1.4 Project uk.ac.york.sesame.testing.generator

We have developed an Eclipse plugin to support the generation of the necessary code for the configurations explored in the SESAME experiments. This can support both the execution of evolutionary experiments, and the generation of fixed custom tests manually defined within the experimental model. This Eclipse plugin is developed in this project. As every Eclipse plugin in order to be able to use it we first need to load it to the registry and this is done by running a new Eclipse Application within the Eclipse IDE. Then the necessary next steps will be explained under Section 4.2.

4.1.5 Project uk.ac.york.sesame.testing.architecture.ros

This project contains an interface to the ROS simulator, which implements ISimulator. The simulation interface uses *jrosbridge*, which provides an interface using ROSbridge upon the MRS side. Additional information about ROS interfacing is contained in Section 4.3.1.

4.1.6 Project uk.ac.york.sesame.testing.architecture.tts

This project exists under the "tts-simulator" branch of the repository, and contains an in-progress interface to the TTS simulator, which implements ISimulator. The simulation interface is implemented using the gRPC protocol [31], and a protobuf protocol definition contained in *SimLogAPI.proto*. This protocol definition is converted into Java code and used in by the TTSSimulator class.

4.2 Code Generation

The user is expected to invoke a new Eclipse Application, under the project uk.ac.york.sesame.testing.generator. This will launch a fresh Eclipse instance, under which they can create a new project. Within this project, they provide an instantiation of the SESAME Testing DSL, specifying the structure of the testing space, as defined in Section 3.5.

In Step 2 of the methodology, metric templates are automatically generated within this plugin project. The testing framework provides a plugin consisting of a wizard with a single page, which can be accessed by right-clicking on the user's newly generated project and selecting "Generate SESAME Code". The plugin provides the option to select the instantiation of the testing DSL. An example of a generated metric template is presented in the right-hand window in the screenshot of Figure 9. The metric template consists of numerous method hooks the user can implement to define the metric initialisation and processing in response to events. In this example, the function *processElement* should be implemented to process incoming MRS events and emit numeric values representing the output of the metric. An example of a completed metric for a case study is presented later in Figure 13.

In order to implement these metrics, the user first needs to copy these classes into a new package `metrics.custom`. Then, it is necessary to implement the needed platform-specific metrics. They need to implement the ProcessElement function if the metric is connected only to a single simulator stream, and both ProcessElement1 and ProcessElement2 functions if the metric is connected to both streams. An example of this metric generation will be provided in the case study section.

4.3 Simulator Interfacing Implementation

4.3.1 ROS Implementation

The ROS simulator models the system components as a number of interacting nodes. Nodes can publish and subscribe to topics in order to communicate with other robotic components. In the ROS implementation, the term topic is used to denote a variable as described in Section 3. In order to support the remapping supplied in Section 3.8, users must provide modified ROS launch scripts, which have been altered in order to contain the remapped names.

On the MRS side, a ROS node must be present in the simulator for *rosbridge*[7], which serves as the logical interfacing module between the simulation-based testing platform and the MRS. Through *rosbridge*, the platform can subscribe to ROS topic updates, triggering notifications when these subscribed topics are updated, and publish new or edit existing topics.

[7] `http://wiki.ros.org/rosbridge_suite`

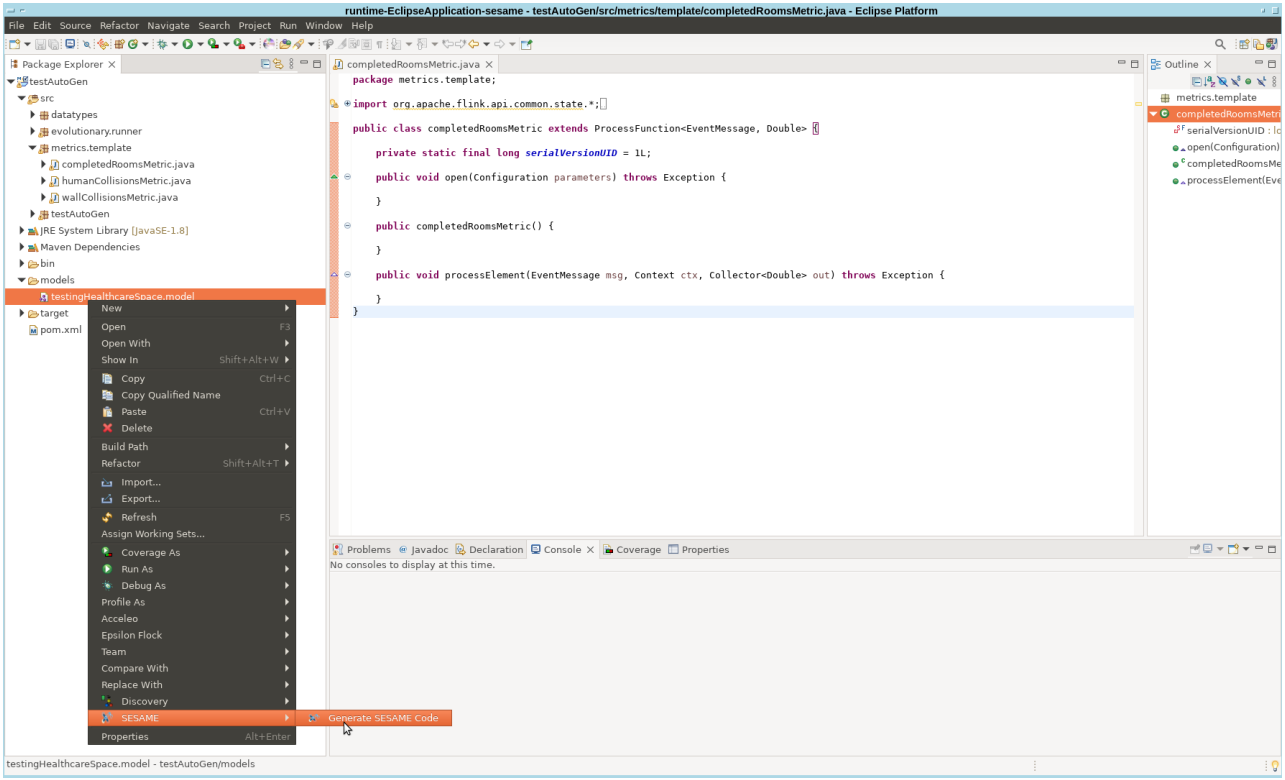


Figure 9: Selection of the SESAME code generation wizard for metric templates. In the generator window on the right, a metric template is produced as its output.

On the middleware side, a ROSSimulator instance is created in the generated code for the test runners, which interfaces with jrosbridge [25] to provide a Java interface to rosbridge. On startup, the middleware configured for a particular test will make subscriptions to the simulator (e.g., to obtain the topics selected for fuzzing and perform the man-in-the-middle approach for variable modification, and any information relevant to monitoring performance metrics to observe safety violations). This information is placed into an Apache Kafka queue named IN, and stream transformations are applied to achieve the man-in-the-middle modifications. The middleware also transfers code from a Kafka OUTPUT queue to transmit modified messages back to the MRS simulator.

4.3.2 TTS Implementation

The TTS simulator-side interface for simulation-based testing is presented as a gRPC protocol server [31] to which our simulation-based testing middleware can connect. The gRPC server-side interface hides the internal implementation of TTS simulator state, providing a ROS message abstraction, so the simulator side can receive standardised ROS messages. This simulator-side interface is also currently being redeveloped by our project partners at TTS in order to provide an abstraction for the KUKA use case, which will hide the details of interfacing to its shared memory, avoiding the need for potentially intrusive and error-prone locking protocols.

The communication between our simulation-based testing framework and the TTS simulator is managed over the gRPC protocol [31]. A protobuf protocol definition (SimLog.API) is presented which can be converted into a Java API by the *protobuf-maven-plugin*. This allows the protocol Java API to be updated dynamically whenever the protocol is updated, which supports convenient development, responding to changes made and ensuring the Java-side interface is consistent with the protocol used by TTS.

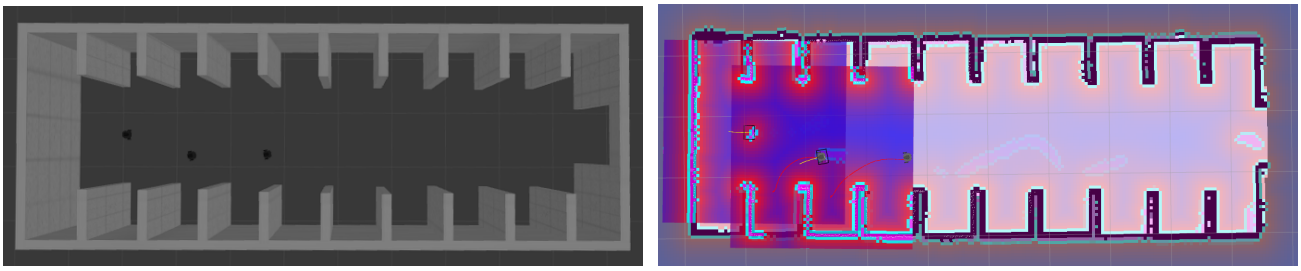
On the middleware side, a class TTSSimulator is presented, as an implementation of the ISimulator interface. Its *connect* method makes a connection to the simulator over a gRPC channel, creating blocking and

non-blocking grpc-Java stubs. The *consumeFromVariable* function creates a custom ROSObserver, which subscribes and is notified when messages arrive from the channel. This ROSObserver connects to the Apache Kafka internal input stream for the middleware, and pushes messages received there into this stream for processing. The *publishToTopic* function creates a new PubRequest, and sets it to contain the given message, before handing it off to be transmitted by the non-blocking stub.

The code for the TTS simulation is currently available in the "tts-simulator" repository branch, in `uk.ac.york.sesame.testing.architecture.tts`. As development continues, the work on this interface will focus on adding time tracking to the TTS simulator, and handling the functionality for automatically starting the simulator from the test runner without manual involvement.

4.3.3 Other Simulator Interfaces

The approach by which the TTS simulator is implemented can be generalised to other simulators. First it is required to implement a simulator-side logical interface, either by interfacing to a simulator plugin API, or implementing it in the source code directly (the latter is an approach we used for MOOS-IvP, adding the ATLASDBInterface component) [34]. Secondly, providing within a middleware package an implementation of the ISimulator interface. This will establish a connection to the simulator logical interface and define functions to consume and publish messages over the appropriate protocol.



(a) Gazebo simulation (b) rviz visualisation

Figure 10: Healthcare case study example showing three robots in operation visiting rooms

5 Case Study

5.1 Overview

This section describes the application and usage of the SESAME simulation-based testing methodology to a simple healthcare-based case study application. The purpose of this section is to show the application of the simulation-based testing methodology in Section 3.2 to this case study, and to present certain parts of the simulation-based testing infrastructure in the context of this case study. This case study uses the same ROS interface as the Locomotec simulator, which should allow us to scale it up to operate with the Locomotec case study when further integration work is completed.

5.2 Case Study Description

5.2.1 Overview

The case study chosen is a team of multi-purpose mobile healthcare robots that collaborate to execute service tasks within a hospital facility. Healthcare robots are increasingly deployed in the clinical domain to carry out routine activities, supporting the often overwhelmed healthcare professionals (i.e., nurses, doctors and other medical staff) in their demanding and fast-paced working environment [35, 45]. The robots can typically perform simple tasks including the delivery of medical materials to care units, transport of medical samples to the lab for analysis, and disinfection of patient rooms and operating suites [36]. Figure 10 shows three Turtlebot 3 Burger robots[8] operating within a simulated healthcare facility comprising a total of 20 rooms (with each room located between the vertical walls and either side of the long horizontal corridor).

Each room may host a single patient or may be empty. When visiting an occupied room, a robot can check whether the patient needs assistance, measure their vital parameters or simply interact with the patient. To support localisation and navigation, each robot is equipped with laser distance and inertia measurement unit sensors (e.g., gyroscope, accelerometer). Figure 10a depicts the simulation environment in Gazebo, whereas Figure 10b shows the rviz visualisation capturing also the localisation area and navigation path per robot (i.e., showing that two robots are moving towards entering the rooms at the bottom left, with the last robot moving to its home location at the left of the hospital corridor).

5.2.2 Case Study Mission Requirements

In this instantiation of the case study, we assume that every robot is pre-assigned to visit a simple sequence of occupied rooms, and to complete a task with a variable completion time within each room. We assume that the

[8] https://www.turtlebot.com

Table 1: Healthcare robots mission requirements

ID	Description
R1	The robots must service all their assigned rooms.
R2	Each robot must return to its starting location for recharging and not become stranded in the hospital environment.
R3	The robots must fulfil requirements R1 and R2 in the shortest possible time and before T_{end} (the maximum mission end time).

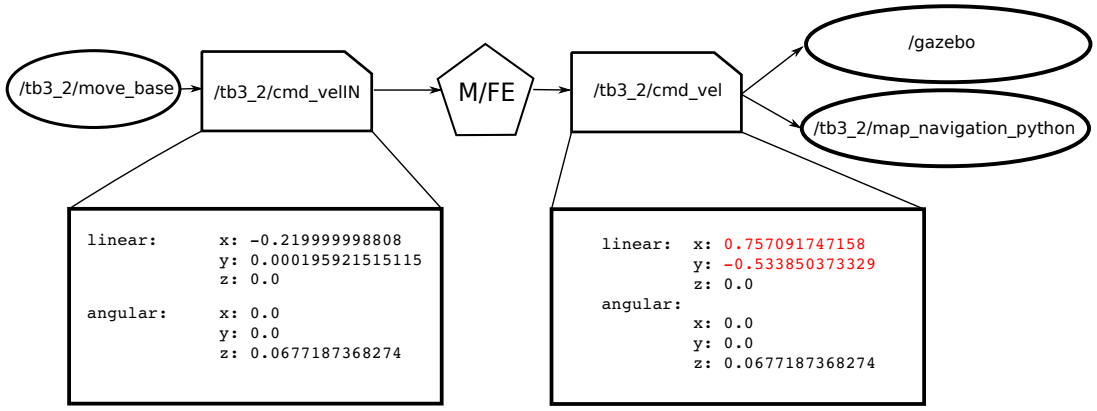


Figure 11: ROS fuzzing message example for case study

factors of primary importance for the case study are robots completing their mission by the defined end time, and returning to their starting location successfully for recharging, without become stranded due to battery exhaustion. The requirements for this case study are as given in Section 1.

5.2.3 Fuzzing Space For Case Study

System testing engineers have identified various fuzzing operations and points in the system which may be impactful in testing this case study. The first fuzzing operation is to attempt to alter the vehicle command velocity from the controller, replacing the intended velocity with a randomised x and y velocity. The impact of these fuzzing operations on the vehicle will be to cause steering errors, either causing robots to leave the intended operating region within the topology, or by causing them to impact with the walls of the hospital environment. Depending on the length of time for which the introduced fuzzing occurs, the locations of the robots at the time, and whether the robots can recover from the injected fuzzing operation, it may not be possible for the robots to continue and recover.

A further fuzzing operation which may be applied to the case study is probabilistic packet loss upon the command velocity topic. This models a case in which there may be (due to electrical disruptions or other interference) transient loss or interruption of messages from the controller. This would cause irregularities in robot control, and perhaps stuttering or overshooting as the robot moves. In both cases, it is possible that these alterations may cause the robot to fail to complete its mission in time, or waste energy, depending on the length of the fuzzing applied and its intensity.

An example of velocity fuzzing upon the command velocity ROS messages for vehicle 2 is shown in Figure 12. In this case the IN side remapping strategy from Section 3.8 is used. The middleware and fuzzing engine listen upon the topic `/tb3_2/cmd_velIN`, and an incoming command velocity message upon this topic from **move_base** is shown. Then, the middleware publishes the fuzzed message with altered linear x and y command velocity values (indicated in red) which is received by the recipient ROS components.

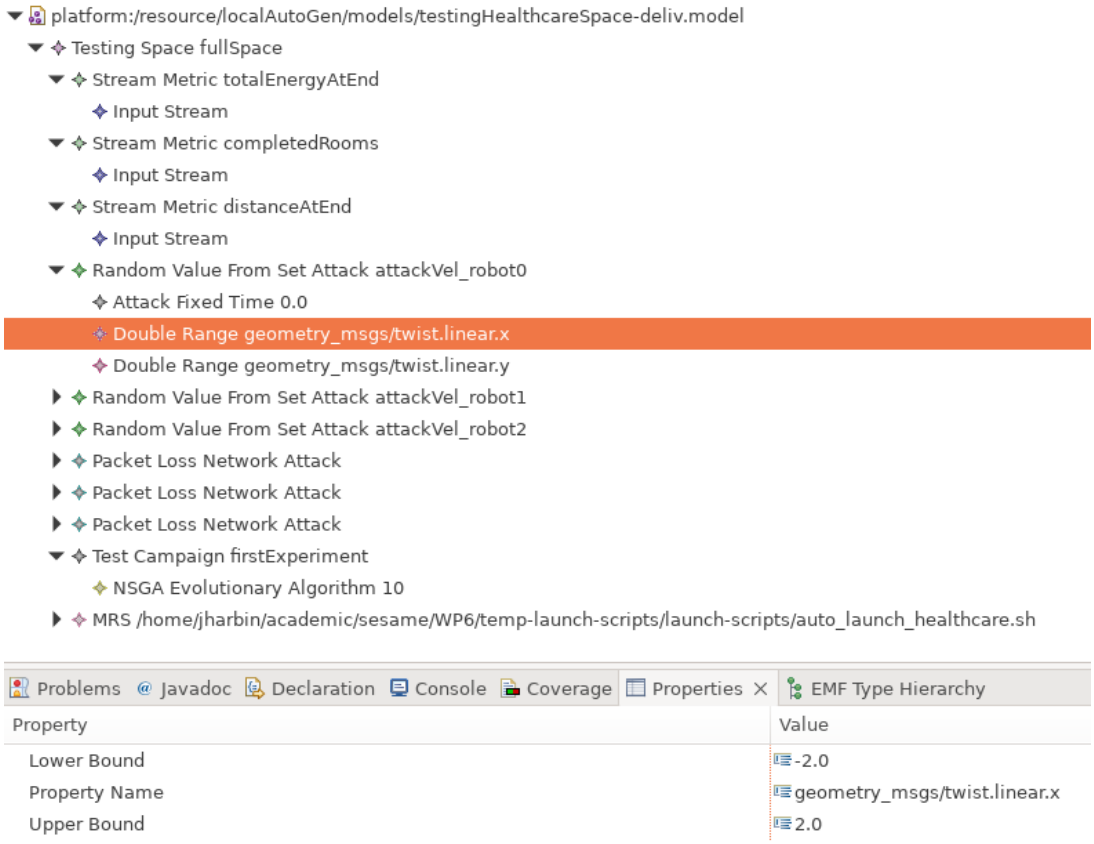


Figure 12: Testing Space Model for Simulation-Based Testing

5.3 Step 1: Model Instantiation

Having specified the potential fuzzing operations that may occur, we will apply the steps from the methodology of Section 3.2 in order to perform simulation-based fuzz testing of the specified case study. In Step 1, an instantiation of the metamodel in Section 3.5 is created in order to model the testing space for fuzz testing specified in Section 5.2.3. The model produced for the case study is shown in the Exceed simulation-based visual editor in Figure 12.

The model specifies the use of three quantitative performance metrics, *totalEnergyAtEnd*, which quantifies the energy that remains at the end of the simulation, *completedRooms*, which gives the number of rooms completed at the end of the simulation, and *distanceAtEnd*, which indicates how far the vehicles are from their intended position at the end. All of these are stream metrics, connected to the simulator IN stream, which means they receive the unfuzzed events from the simulator. A further metric which is not based on simulator state is supplied, *fuzzingTimeLength*, that sums the total time length of attacks in the model, and returns this static value as a stream. This allows the users to minimise the total fuzzing time length, thereby favouring the more impactful but shorter fuzzing operations.

In regards to the fuzzing operations selected to make up the complete fuzzing space in Section 5.2.3, the RandomValueFromSetOperation is chosen, which is repeated three times (one per each of the three vehicles in the case study). The specific properties in the messages to be adjusted are selected by specifying ValueSets with parameter names geometry_msgs/twist.linear.x and geometry_msgs/twist.linear.y. For each ValueSet, the boundaries of each are set from -1.0 to 1.0, which define in metres per second the range of the fuzzed parameter injected.

The probabilistic message dropping by fuzzing operations is applied using a PacketLossNetworkOperation which represents potentially dropping the message values in transit. The probability range for dropping is

defined as 0.0 to 1.0, which indicates that individual fuzzing operations in generated tests can be specialised anywhere in the full probability range. Again, three copies of the fuzzing operation are included in the model, one selected to disrupt each of the command velocity topics for the different robots.

5.4 Step 2: Code Generation

Code generation can be used to generate metric templates automatically, within the newly generated project under the child Eclipse instance. The simulation-based testing framework provides a plugin consisting of a wizard with a single page, which can be accessed by right-clicking on the user's newly generated project and selecting "Generate SESAME Code". The plugin provides the option to select the instantiated Testing DSL.

5.5 Step 3: Metric Development

The next step is for the user to define scenario-specific performance metrics, in order to quantify violations of the mission requirements specified in Table 1. In order to implement the metrics, the user first needs to copy these classes into a new package `metrics.custom`. Users then need to implement the needed platform-specific metrics as Java code. They need to implement the processElement stub function If the metric is connected only to a single simulator stream, and both processElement1 and processElement2 functions if their metrics are connected to both streams.

5.6 Metric Implementation

Figure 13 gives an example metric which has been implemented for the requirement *totalCompletedRooms*, in order to track the number of successfully completed rooms within a robotic mission. It listens to incoming simulator events on the simulator IN stream (receiving messages from the MRS) and reacts to messages from the MRS on any robot's topic roomCompleted, whose value contains the ID of a newly completed room. The metric responds by incrementing its room completion counter. The total number of completed rooms at that instant is emitted as the metric's output, which can be logged as a time series by the simulation-based testing interface. The final value is also stored in the DSL for this particular Test MetricInstance, and can be used as an input to a multi-objective optimisation process:

5.7 Step 4: Experiment Execution

A single experiment is defined, with the metrics selected to maximise the number of completed rooms. The experiment is defined to use the NSGA-II algorithm. The chosen population size is set to 10, and the number of iterations is set to 200.

The user should create an Eclipse Run Configuration for the ExptRuner_name.java for the experiment they would like to execute, and invoke this Run Configuration to run the experiment. This runner will be configured with the parameters chosen from the testing DSL.

The experiment runner will generate tests according to the strategy specified for the experiment, for example, with NSGA-II it will perform multi-objective optimisation using this genetic algorithm A population of Tests is generated and evaluated. Each test is evaluated by first dynamically generating a specialised test runner which acts as a middleware interfacing with the low-level MRS and modifying its internal messages, using any custom supplied metric definitions provided in Step 2 to quantify the impact of the fuzzing test. The interfacing to the ROS simulation is performed using *rosbridge* on the ROS topology side, and *jrosbridge* under the Java side of the simulation-based testing platform, under the ROSSimulator class.

```
package metrics.custom;

import org.apache.flink.api.common.state.*;
import org.apache.flink.configuration.Configuration;
import org.apache.flink.streaming.api.functions.*;
import org.apache.flink.util.Collector;

import uk.ac.york.sesame.testing.architecture.data.EventMessage;

public class completedRoomsMetric extends ProcessFunction<EventMessage, Double> {

        private static final long serialVersionUID = 1L;
        private ValueState<Long> totalRoomsCompleted;

    public void open(Configuration parameters) throws Exception {
        totalRoomsCompleted = getRuntimeContext().
            getState(new ValueStateDescriptor<>("totalRoomsCompleted", Long.class));
    }

    public void processElement(EventMessage msg, Context ctx, Collector<Double> out) throws Exception {
        String completionTopicName = "roomCompleted";
        if (msg.getTopic().contains(completionTopicName)) {
                Object v = msg.getValue();
                totalRoomsCompleted.update(totalRoomsCompleted.value() + 1);
        }

        out.collect(Double.valueOf(totalRoomsCompleted.value()));
    }
}
```

Figure 13: Metric implementation example for healthcare case study - completedRoomsMetric

The metric information from the low-level test runner is communicated to the experiment runner and used to guide a multi-objective optimisation process. This process uses genetic operations such as mutation and crossover to create new tests, discarding the worst performing campaigns from the population. The metric results from the experiment are logged back to the DSL as MetricInstances under the performed Tests in the DSL.

5.7.1 Fuzzing Operation Code Generation

As part of the generated code produced automatically during the experiment, several fuzzing operations are generated and included within the generated code for particular test runners, in the root package. Depending on the configuration of a particular test, these fuzzing operations may be activated and installed within a chain of Flink flatmaps in order to transform the events in the system. An example of the fuzzing operation which is employed in order to modify and transform the velocity values on robot ID 0 with the RandomValueFromSet-Operation is given in Figure 14. The message transformation is performed in the *flatMap* method, which uses JSON utility methods in the middleware to transform and replace the part of the contents of a particular structured JSON message. The generated code is set to operate upon the message fields specified in the DSL, so it replaced the x and y velocity values within the ranges permitted for this test.

An additional fuzzing operation class generated is PacketLossNetworkOperation. This class uses a random generator to determine when messages which are matching its criteria should be forwarded or not, depending on the frequency parameter supplied in the test runner for the fuzzing operation. Every time a message upon a matching topic is received (and the fuzzing operation is active) it may optionally discard the message based on the random number generated. The code for this is presented in Figure 15.

```
import java.util.Map;
import java.util.HashMap;
import java.util.Random;
import datatypes.DoubleRange;

import org.apache.flink.util.Collector;
import org.json.simple.*;

import uk.ac.york.sesame.testing.architecture.utilities.ParsingUtils;
import uk.ac.york.sesame.testing.architecture.fuzzingoperations.FuzzingOperations;
import uk.ac.york.sesame.testing.architecture.data.EventMessage;

public class fuzzingopVel_robot0FlatMap_Test_003_29_06_2022_23_15_33 extends FuzzingOperations {

        private static final long serialVersionUID = 1L;
        Random rng;

        public fuzzingopVel_robot0FlatMap_Test_003_29_06_2022_23_15_33
                (String topic, String start, String end, long seed) {
                super(topic, start,end);
                this.rng = new Random(seed);
        }

        @Override
        public void flatMap(EventMessage value, Collector<EventMessage> out) throws Exception {
          if (value.getTopic().equals(topic) && isReadyNow()) {
            Object obj = JSONValue.parse(value.getValue().toString());
            JSONObject jo = (JSONObject)obj;
            jo = ParsingUtils.updateJSONObject(jo, "geometry_msgs/twist.linear.x",
                new DoubleRange(-0.6882,-0.1577).generateInRange(rng));
            jo = ParsingUtils.updateJSONObject(jo, "geometry_msgs/twist.linear.y",
                new DoubleRange(-0.5820, 0.7594).generateInRange(rng));
            EventMessage valueOut = new EventMessage(value);
            valueOut.setValue(jo.toString());
            out.collect(valueOut);
          } else {
            out.collect(value);
          }
        }
}

```

Figure 14: Auto-generated fuzzing operation code for RandomValueFromSetOperation for case study

5.8 Case Study Test Experiment

An experimental evaluation was performed for the case study described in this section. Three metrics were selected for this experiment; *completedRooms*, i.e., the number of rooms successfully serviced by the robots, *distanceAtEnd*, i.e., the total distance of robots from their starting point at the end of the simulation, and *fuzzingTimeLength*, i.e., the total time length of fuzzing operations in a particular test.

The optimisation of selected metrics was set to minimise *completedRooms*, maximise *distanceAtEnd*, and minimise the *fuzzingTimeLength*. This combination prefers scenarios that create the maximum number of violations of requirements R1 and R2, (i.e., robots failing to service their assigned rooms, and becoming stranded in the hospital environment due to the effects of fuzzing). It also prefers tests that minimise the total fuzzing length in the process, thereby finding the most impactful but shortest fuzzing operations in the process.

The experimental time configured for execution of the scenario was 300 seconds, although startup delays for initiating the ROS processes and Python scripts comprising the MRS made the experiment last approximately 280 seconds. The scenario as intended involves a total of nine rooms assigned for the robots to complete. This provides sufficient time (in the absence of fuzzing) for the robots to complete their mission and to return to the starting points for recharging.

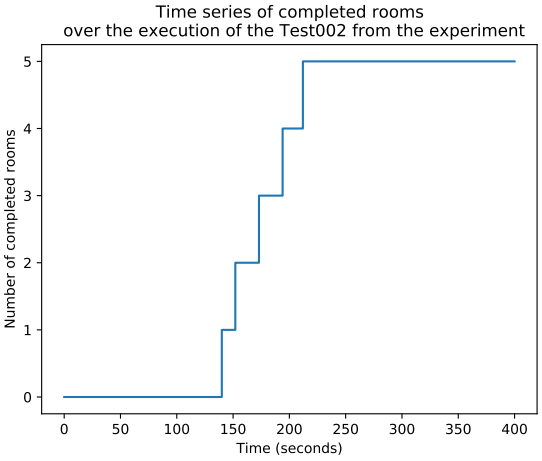
```
import org.apache.flink.util.Collector;
import java.util.Random;

import uk.ac.york.sesame.testing.architecture.fuzzingoperations.FuzzingOperation;
import uk.ac.york.sesame.testing.architecture.data.EventMessage;

public class PacketLossFlatMap_Test_014_29_06_2022_23_59_47 extends FuzzingOperation {
        private static final long serialVersionUID = 1L;
        double frequency;

        public double getFrequency() {
                return frequency;
        }

        public void setFrequency(double frequency) {
                this.frequency = frequency;
        }

        public PacketLossFlatMap_Test_014_29_06_2022_23_59_47
                (String topic, String start, String end, double frequency) {
                super(topic, start, end);
                this.frequency = frequency;
        }

        @Override
        public void flatMap(EventMessage value, Collector<EventMessage> out) throws Exception {
                Random rand = new Random();
                System.out.println("frequency = " + frequency);
                if (value.getTopic().equalsIgnoreCase(topic) && isReadyNow()) {
                  if (frequency <= rand.nextDouble()) {
                    System.out.println("DISCARDING: Message: " + value + " discarded.");
                  } else {
                    System.out.println("ALLOWING: Message " + value);
                    out.collect(value);
                  }
                } else {
                  System.out.println("ALLOWING: Message " + value);
                  out.collect(value);
                }
        }
}
```

Figure 15: Auto-generated fuzzing operation code for PacketLossFuzzingOperation for case study

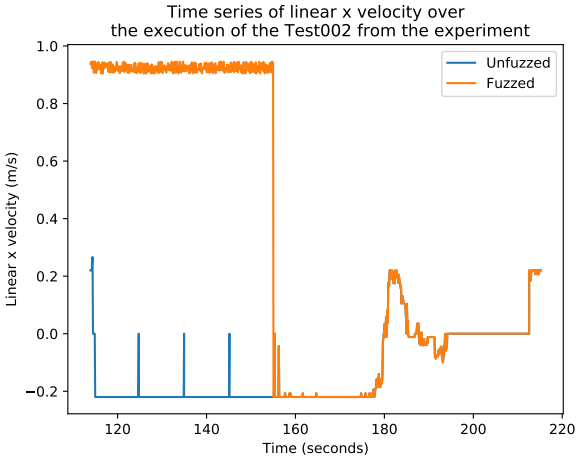
5.8.1 Fuzzing Results for Example Test Configuration

Time series plots from an example test result (i.e., Test002) found from the output population of the GA is illustrated in Figure 16. This test consists of a velocity fuzzing operation in robot 2's command velocity topic, similar to that in Section 5.2.3. The test configuration randomises the linear x and y velocity such that $0.904 < x < 0.946$, and $-0.648 < y < -0.012$. The time of activation (as measured from the **/clock** topic) is between 101.6 seconds and 154.9 seconds. Since in this MRS system the **/clock** topic, generated for the MRS from Gazebo, has a starting time offset of 100 seconds and there is a delay of around 10 seconds from this for all MRS components to start up, effectively the fuzzing is active immediately upon system startup.

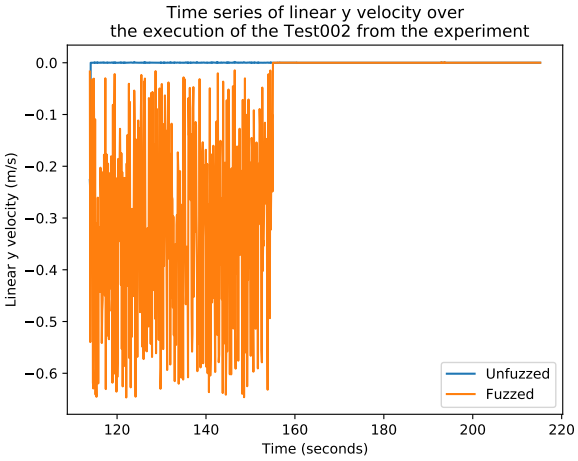
The *completedRooms* metric over time is shown in Figure 16a. This illustrates that Test002, as a result of the fuzzing applied, completed only 5 rooms out of the 9 assigned to the robots. The activation of the fuzzing operations is visible in Figures 16b and 16c, which compares the x and y velocity values (fuzzed and unfuzzed), captured using *rostopic* for the fuzzing test case. In this case, the x fuzzing operation selected had a very narrow range as can be seen from the fuzzed x data series. The fuzzed y values here were generated with a large negative range, as illustrated in the fuzzed data series. It is possible to activate the test runner for Test002 manually and inspect its behaviour upon the Gazebo GUI. This process shows that on the activation of the test case, the fuzzing immediately activates and drives robot 2 into the wall of one of the starting points. On the end of the fuzzing activation, approximately 50 seconds later, robot 2 returns to its intended behaviour pattern, and depending on variations in the timing, may make a close approach or collision with another robot,



(a) Time series of *completedRooms* metric



(b) Time series of fuzzed and unfuzzed x velocity



(c) Time series of fuzzed and unfuzzed y velocity

Figure 16: Results from an example fuzzing test case (Test002)

delaying its mission execution. Therefore, the completed rooms metric of 5 likely resulted from 3 missing room completions for robot 2, and 1 for another robot as a result of this collision.

5.8.2 Lessons Learned

The navigation between robots occasionally become confused when navigating between adjacent rooms in the healthcare case study, especially if a momentary fuzzing error drives a robot into or close to the wall, or close to another robot. In this case, the robotic navigation seems to repeatedly reverse and drive forwards, before giving up and does not restore control. Therefore, even sometimes when a fuzzing error is transient, the mission may exhibit large deviations from a benign setting. This provides evidence for the high sensitivity of the robotic navigation stack. Also, sometimes a relatively short glancing impact into a wall can stall the vehicle, and it does not recover normal control. Therefore, components such as amcl and move_base may not be sufficiently stable to use for this type of control in situations in which this type of errors exist (e.g., due to environmental noise).

A related additional safety issue detected with the case study not captured by our formal fuzzing requirements was a collision upon starting with two closely spaced robots. A relatively short burst of fuzzing upon the command velocity topic for either of these could lead to the two vehicles colliding with each other. Although

this would be frequently corrected and the mission resume normally, it could in a real implementation lead to hazards or damage one of the vehicles, especially when other humans are involved closely in the healthcare area. Therefore, recharging points for multiple robots should be spaced more widely in order to provide additional distance at these vulnerable points in the simulation.

6 Conclusion

This report has presented the various strategies that are used to meet the challenges that are presented for robotic simulation-based testing in Section 1.2.

We propose the use of fuzz testing in order to provide repeatable disruptions to MRS simulations, together with a flexible requirement monitoring which can be customised by the user in order to monitor custom scenario-specific metrics. By repeatedly executing particular scenarios from generated tests, it is possible to identify the disruptions that occurred in hard-to-reproduce events, and examination of available simulator log files in these circumstances enables test engineers to reconstruct the events which occurred.

We propose a testing DSL in order to specify the full space of possible fuzzing operations that can be applied to a given robotic system, while allowing system developers to constrain the available fuzzing operations and focus upon the most interesting regions of the search space within the limited time available for experiments. Compared to existing work, the SESAME simulation-based testing platform is one of the first model-driven fuzzing approaches for the robotics domain. Its main focus is to identify high-level system design faults instead of low-level faults in the MRS. The simulation-based testing platform raises the level of abstraction and provides simulator independence, by providing a generic simulator interface. The simulator is provided for ROS/Gazebo and an in progress interface is available for the TTS simulator for the KUKA use case.

The simulation-based testing platform provides an automated approach for exploring the testing space. Currently, the platform supports the dynamic generation and execution of new tests, using feedback from scenario-specific information to guide a multi-objective optimisation loop. This allows the platform to focus on the most interesting regions in the testing space, which produce the most violations. Finally, we have discussed how the platform can interface with EDDIs, and how EDDIs can inform simulation-based testing, as well as how the platform can support testing of the EDDIs' behaviour.

7 Satisfaction of SESAME Requirements

Table 2: Satisfaction of SESAME Requirements extracted from D1.1: Project Requirements

Priority	Requirement	Status
Shall	Support semi-automated transformation of ExSce definition into testing scenarios to be exercised in simulation	Partial: placeholder executable scenarios can be transformed into testing cases, with variations in the fuzz testing applied and automatically simulated
Shall	Support the execution of simulated ExSce with tracking of acceptance criteria and provenance data in the form of logs	Full: the simulated scenarios can be executed to determine their metric values according to scenario-specific metrics. Metrics are logged to the DSL to record their success
Shall	Define an open API specification for the monitoring and manipulation of simulated ExSce at runtime	This will be completed as part of Task 6.5
Shall	Support data stream acquisition from the real devices into a Digital Twin platform	This will be completed as part of Task 6.4
Shall	Support integration of simulated ExSce with EDDI logic with bi-directional data exchange	This will be completed as part of Task 6.5
Shall	Support the interpretation of ExSce	Partial: we can extract information from a placeholder for the ExSces and use to drive the test case generation
Should	Support integration of different simulation platforms	Partial: the platform supports the ROS simulator and is being extended for TTS simulator interfacing
Should	Support simulation model updating according to data stream analysis on the Digital Twin Platform	This will be completed as part of Task 6.5
May	Support Virtual Commissioning mode for hardware in the loop analysis	This will be investigated as part of Task 6.4
May	Provide a configuration User Interface to ease the Model to Model transformation	Partial: we have automated code generation as part of an early plugin wizard, which will be extended and provided in the next version of the testing tool.
May	Support interfacing with EDDI	In progress

References

[1] Afsoon Afzal, Claire Le Goues, Michael Hilton, and Christopher Steven Timperley. A study on challenges of testing robotic systems. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 96–107, 2020.

[2] Afsoon Afzal, Claire Le Goues, Michael Hilton, and Christopher Steven Timperley. A study on challenges of testing robotic systems. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 96–107. IEEE, 2020.

[3] Andreas Zeller. Fuzzing: A tale of two cultures, 2022.

[4] OpenAI: Marcin Andrychowicz, Bowen Baker, Maciek Chociej, Rafal Jozefowicz, Bob McGrew, Jakub Pachocki, Arthur Petron, Matthias Plappert, Glenn Powell, Alex Ray, et al. Learning dexterous in-hand manipulation. *The International Journal of Robotics Research*, 39(1):3–20, 2020.

[5] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. Nautilus: Fishing for deep bugs with grammars. In *NDSS*, 2019.

[6] Philipp A Baer, Roland Reichle, and Kurt Geihs. The spica development framework–model-driven software development for autonomous mobile robots. In *Proceedings of the 10th international conference on intelligent autonomous systems (IAS-10'08)*, pages 211–220, 2008.

[7] Raja Ben Abdessalem, Shiva Nejati, Lionel C Briand, and Thomas Stifter. Testing advanced driver assistance systems using multi-objective search and neural networks. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, pages 63–74, 2016.

[8] M. Benjamin, H. Schmidt, P. Newman, and J. Leonard. *Autonomy for unmanned marine vehicles with MOOS-IvP*, pages 47–90. Springer, 2013.

[9] Jonathan Bohren and Steve Cousins. The smach high-level executive [ros news]. *IEEE Robotics & Automation Magazine*, 17(4):18–20, 2010.

[10] Darko Bozhinoski, Davide Di Ruscio, Ivano Malavolta, Patrizio Pelliccione, and Ivica Crnkovic. Safety for mobile robotic systems: A systematic mapping study from a software engineering perspective. *Journal of Systems and Software*, 151:150–179, 2019.

[11] Herman Bruyninckx, Markus Klotzbücher, Nico Hochgeschwender, Gerhard Kraetzschmar, Luca Gherardi, and Davide Brugali. The brics component model: a model-based development paradigm for complex robotics software systems. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1758–1764, 2013.

[12] Giuseppina Lucia Casalaro, Giulio Cattivera, Federico Ciccozzi, Ivano Malavolta, Andreas Wortmann, and Patrizio Pelliccione. Model-driven engineering for mobile robotic systems: a systematic mapping study. *Software and Systems Modeling*, pages 1–31, 2021.

[13] G Cattivera and GL Casalaro. Model-driven engineering for mobile robot systems: A systematic mapping study. *Malardalen University*, 2015.

[14] Yevgen Chebotar, Ankur Handa, Viktor Makoviychuk, Miles Macklin, Jan Issac, Nathan Ratliff, and Dieter Fox. Closing the sim-to-real loop: Adapting simulation randomization with real world experience. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 8973–8979. IEEE, 2019.

[15] Federico Ciccozzi, Davide Di Ruscio, Ivano Malavolta, and Patrizio Pelliccione. Adopting mde for specifying and executing civilian missions of mobile multi-robot systems. *IEEE Access*, 4:6451–6466, 2016.

[16] Federico Ciccozzi, Davide Di Ruscio, Ivano Malavolta, Patrizio Pelliccione, and Jana Tumova. Engineering the software of robotic systems. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 507–508. IEEE, 2017.

[17] Carlos A Coello Coello, Gary B Lamont, David A Van Veldhuizen, et al. *Evolutionary algorithms for solving multi-objective problems*, volume 5. Springer, 2007.

[18] Edson de Araújo Silva, Eduardo Valentin, Jose Reginaldo Hughes Carvalho, and Raimundo da Silva Barreto. A survey of model driven engineering in robotics. *Journal of Computer Languages*, page 101021, 2021.

[19] Rogério De Lemos et al. Software engineering for self-adaptive systems: A second research roadmap. In *Software Engineering for Self-Adaptive Systems II*, pages 1–32. Springer, 2013.

[20] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[21] Rodrigo Delgado, Miguel Campusano, and Alexandre Bergel. Fuzz testing in behavior-based robotics. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 9375–9381. IEEE, 2021.

[22] Saadia Dhouib, Selma Kchir, Serge Stinckwich, Tewfik Ziadi, and Mikal Ziane. Robotml, a domain-specific language to design, simulate and deploy robotic applications. In *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, pages 149–160. Springer, 2012.

[23] Clare Dixon, Alan Winfield, and Michael Fisher. Towards temporal verification of emergent behaviours in swarm robotic systems. In Roderich Groß, Lyuba Alboul, Chris Melhuish, Mark Witkowski, Tony J. Prescott, and Jacques Penders, editors, *Towards Autonomous Robotic Systems*, pages 336–347, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[24] Swaib Dragule, Bart Meyers, and Patrizio Pelliccione. A generated property specification language for resilient multirobot missions. In *International Workshop on Software Engineering for Resilient Systems*, pages 45–61. Springer, 2017.

[25] Russel Toris et al. jrosbridge: A Native Java EE rosbridge Client. `https://github.com/rctoris/jrosbridge`, 2021. last accessed: 2021-06-20.

[26] José M Gascuena, Elena Navarro, and Antonio Fernández-Caballero. Model-driven engineering techniques for the development of multi-agent systems. *Engineering Applications of Artificial Intelligence*, 25(1):159–173, 2012.

[27] Gazebo (2018): Robot simulator made easy, 2018.

[28] Simos Gerasimou, Nicholas Matragkas, and Radu Calinescu. Towards systematic engineering of collaborative heterogeneous robotic systems. In *2019 IEEE/ACM 2nd International Workshop on Robotics Software Engineering*, pages 25–28. IEEE, 2019.

[29] Mario Gleirscher, Simon Foster, and Jim Woodcock. New opportunities for integrated formal methods. *ACM Computing Surveys (CSUR)*, 52(6):1–36, 2019.

[30] Patrice Godefroid. Fuzzing: Hack, art, and science. *Communications of the ACM*, 63(2):70–76, 2020.

[31] Louis Ryan (Google). grpc motivation and design principles. =https://grpc.io/blog/principles/. Accessed: 2022-06-28.

[32] James Harbin, Simos Gerasimou, and Nicholas Matragkas. Testing robotic systems using model-based evolutionary-guided fuzzing (submitted). In *25th International Conference on Model Driven Engineering Languages and Systems (MODELS' 22)*. IEEE, Oct 2022.

[33] James Harbin, Simos Gerasimou, Nicholas Matragkas, Athanasios Zolotas, and Radu Calinescu. Model-driven simulation-based analysis for multi-robot systems. In *24th International Conference on Model Driven Engineering Languages and Systems (MODELS' 21)*. IEEE, Oct 2021.

[34] James Harbin, Simos Gerasimou, Nicholas Matragkas, Athanasios Zolotas, and Radu Calinescu. Model-driven simulation-based analysis for multi-robot systems. In *2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 331–341. IEEE, 2021.

[35] Nick Hawes, Christopher Burbridge, Ferdian Jovan, Lars Kunze, Bruno Lacerda, Lenka Mudrova, Jay Young, Jeremy Wyatt, Denise Hebesberger, Tobias Kortner, et al. The strands project: Long-term autonomy in everyday environments. *IEEE Robotics & Automation Magazine*, 24(3):146–156, 2017.

[36] Denise Hebesberger, Tobias Körtner, Jürgen Pripfl, Christoph Gisinger, Marc Hanheide, et al. What do staff in eldercare want a robot for? an assessment of potential tasks and user requirements for a long-term deployment. In *IROS Workshop on "Bridging user needs to deployed applications of service robots"*, 2015.

[37] Tom P Huck, Christoph Ledermann, and Torsten Kröger. Simulation-based testing for early safety-validation of robot systems. In *2020 IEEE Symposium on Product Compliance Engineering-(SPCE Portland)*, pages 1–6. IEEE, 2020.

[38] Arthur Juliani, Vincent-Pierre Berges, Ervin Teng, Andrew Cohen, Jonathan Harper, Chris Elion, Chris Goy, Yuan Gao, Hunter Henry, Marwan Mattar, et al. Unity: A general platform for intelligent agents. *arXiv preprint arXiv:1809.02627*, 2018.

[39] Neeraj Karamchandani, Vinay Sachidananda, Suhas Setikere, Jianying Zhou, and Yuval Elovici. Smuf: State machine based mutational fuzzing framework for internet of things. In *International Conference on Critical Information Infrastructures Security*, pages 101–112. Springer, 2018.

[40] Francisco Martínez Lasaca and Gijs van der Hoorn. Automatic fuzzing for ros 2. `https://github.com/rosin-project/ros2_fuzz`, 2022.

[41] Alvaro Miyazawa, Pedro Ribeiro, Wei Li, Ana Cavalcanti, Jon Timmis, and Jim Woodcock. Robochart: modelling and verification of the functional behaviour of robotic applications. *Software & Systems Modeling*, 18(5):3097–3149, 2019.

[42] David Molnar, Xue Cong Li, and David A Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *USENIX Security Symposium*, volume 9, pages 67–82, 2009.

[43] Hoang Lam Nguyen, Nebras Nassar, Timo Kehrer, and Lars Grunske. Mofuzz: A fuzzer suite for testing model-driven software engineering tools. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1103–1115. IEEE, 2020.

[44] Arne Nordmann, Nico Hochgeschwender, Dennis Wigand, and Sebastian Wrede. A survey on domain-specific modeling and languages in robotics. *Journal of Software Engineering in Robotics*, 7(1):75–99, 2016.

[45] A. M. Okamura, M. J. Mataric, and H. I. Christensen. Medical and health-care robotics. *IEEE Robotics Automation Magazine*, 17(3):26–37, Sep. 2010.

[46] Michał H Pałka, Koen Claessen, Alejandro Russo, and John Hughes. Testing an optimising compiler by generating random lambda terms. In *Proceedings of the 6th International Workshop on Automation of Software Test*, pages 91–97, 2011.

[47] Alexandros Paraschos, Nikolaos I Spanoudakis, and Michail G Lagoudakis. Model-driven behavior specification for robotic teams. In *AAMAS*, pages 171–178, 2012.

[48] SESAME Project Partners. Sesame evaluation plan. Technical report, The Open Group, Sep 2021.

[49] SESAME Project Partners. Runtime safety and security concept – executable digital dependability identity (eddi) runtime model specification. Technical report, The Open Group, Jun 2022.

[50] SESAME Project Partners. Safety-security co-engineering framework. Technical report, The Open Group, Jun 2022.

[51] Carlo Pinciroli and Giovanni Beltrame. Buzz: An extensible programming language for heterogeneous swarm robotics. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3794–3800. IEEE, 2016.

[52] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Ng. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*, volume 3, 01 2009.

[53] Sean Rivera, Antonio Ken Iannillo, et al. Discofuzzer: Discontinuity-based vulnerability detector for robotic systems, 2020.

[54] Christian Schlegel, Thomas Haßler, Alex Lotz, and Andreas Steck. Robotic software systems: From code-driven to model-driven designs. In *2009 International Conference on Advanced Robotics*, pages 1–8. IEEE, 2009.

[55] Christian Schlegel, Andreas Steck, Davide Brugali, and Alois Knoll. Design abstraction and processes in robotics: From code-driven to model-driven engineering. In *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, pages 324–335. Springer, 2010.

[56] Martin Schneider, Jürgen Großmann, Ina Schieferdecker, and Andrej Pietschker. Online model-based behavioral fuzzing. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, pages 469–475. IEEE, 2013.

[57] Radu Serban, Michael Taylor, Dan Negrut, and Alessandro Tasora. Chrono:: Vehicle: template-based ground vehicle modelling and simulation. *International Journal of Vehicle Performance*, 5(1):18–39, 2019.

[58] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and Service Robotics*, 2017.

[59] Hendrik Skubch, Michael Wagner, Roland Reichle, and Kurt Geihs. A modelling language for cooperative plans in highly dynamic domains. *Mechatronics*, 21(2):423–433, 2011.

[60] Thierry Sotiropoulos, Hélene Waeselynck, Jérémie Guiochet, and Félix Ingrand. Can robot navigation bugs be found in simulation? an exploratory study. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 150–159. IEEE, 2017.

[61] SAFEMUV Project Team. Safemuv: Safe airframe inspection using multiple uavs. Technical report, University of York, Jan 2022.

[62] Christopher Steven Timperley, Afsoon Afzal, Deborah S Katz, Jam Marcos Hernandez, and Claire Le Goues. Crashing simulated planes is cheap: Can simulation detect robotics bugs early? In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 331–342. IEEE, 2018.

[63] Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. In *2017 IEEE/RSJ international conference on intelligent robots and systems (IROS)*, pages 23–30. IEEE, 2017.

[64] Vasudev Vikram, Rohan Padhye, and Koushik Sen. Growing a test corpus with bonsai fuzzing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 723–735. IEEE, 2021.

[65] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 724–735, 2019.

[66] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *2010 IEEE Symposium on Security and Privacy*, pages 497–512. IEEE, 2010.

[67] Trey Woodlief, Sebastian Elbaum, and Kevin Sullivan. Fuzzing mobile robot environments for fast automated crash detection. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5417–5423. IEEE, 2021.

[68] Michał Zalewski. Technical "whitepaper" for afl-fuzz, 2013.

[69] Allan Zhao, Jie Xu, Mina Konaković-Luković, Josephine Hughes, Andrew Spielberg, Daniela Rus, and Wojciech Matusik. Robogrammar: graph grammar for terrain-optimized robot design. *ACM Transactions on Graphics (TOG)*, 39(6):1–16, 2020.

[70] Ding Zhao and Huei Peng. From the lab to the street: Solving the challenge of accelerating automated vehicle testing. *ArXiv*, abs/1707.04792, 2017.