THE FRAMEWORK PROGRAMME FOR RESEARCH AND INNOVATION
HORIZON 2020

SESAME
SECURE AND SAFE MULTI-ROBOT SYSTEMS

Project Number 101017258

D6.6 Multi-Stage Quality Assurance Methodology for EDDI-Supported MRS

Version 1.0
7 July 2023
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 Panayiotis Kolios 1 Panepistimiou Avenue 2109 Aglatzia, Nicosia Cyprus E-mail: kolios.panayiotis@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	2 May 2023
0.2	Initial draft	2 June 2023
0.3	Completed draft	16 June 2023
0.4	Completed draft	23 June 2023
0.8	Revision after reviews	29 June 2023
1.0	Completed QA version	7 July 2023

Table of Contents

List of Figures

Executive Summary

This document presents the integrated methodology for the transition between simulation-based testing and physical testing of MRS systems. It also presents the final version of the simulation-based testing component of the SESAME platform, incorporating changes from our previous deliverable SESAME D6.2. It focuses on the techniques and tools developed for Task 6.5:

Task 6.5: Development of a Multi-Staged Quality Assurance Methodology for EDDI-Supported MRS

A methodology for integrated testing and managing the transition between simulation-based testing and lab experimentation is presented. Following a specification of the scenario requirements, custom performance metrics and an EDDI, the chosen scenario can be explored with simulation-based testing. We present two approaches to fuzzing in this version of the platform, this time supporting condition-based fuzzing as well as time-based fuzzing. We used a DSL to allow test engineers and system integrators to define the space of operations, and the parameters of the experiments to be performed.

Our tool incorporates an evolutionary experiment runner which consumes experimental test campaign definitions, and defines and explores new tests dynamically. Scenario-specific performance metrics are used to assess the quality of the configurations discovered in terms of exposing violations of scenario requirements. The exploration can be guided by a novel strategy for assessing the coverage of a reduced space of potential fuzzing configurations, together with a coverage-boosting technique that seeks to increase diversity in the fuzzing configurations explored. This is performed by dimensionality reduction that reduces each fuzzing configuration to a point in configuration space, together with ensuring sufficient occupation as a termination condition of the evolution. Increased diversity is provided by modifications to the mutation operators.

Following this, we present an algorithm to select a subset of the output configurations discovered in evolution for physical testing, allowing test engineers to use a hyper-parameter to manage the tradeoff between exploitation of the best results (in terms of their violations discovered) and exploration of the parameter space (to find potentially widely spaced reality gaps). Consideration is given to how safety can be ensured in physical testing, and how to proceed in the case of reality gaps identified. An application of our methodology is presented using the KUKA industrial assembly case study with the TTS simulator, presenting an example EDDI, the models and example testing campaign.

Structure of the Document

- Section 1 provides an introduction to the challenges of MRS 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 a summary of the overall integrated overall methodology and its first phases incorporating our simulation-based testing approach
- Section 4 presents the later phases of the methodology, incorporating the transition from simulation-based to physical testing.
- Section 5 presents specific details about its technical implementation
- Section 6 presents the application of our tool to an industrial case study example on the KUKA use case
- Section 7 concludes the report

1 Introduction

This document summarises the activities concerning transitioning from simulation-based testing to physical testing of MRS robotic systems. Since simulation results rely upon assumptions about the real world, physical testing in the lab and operational environment is still necessary in order to verify the correctness of these assumptions. This task has explored a systematic and model-driven approach that enables the identification and selection of dependability-critical scenarios and their progressive evaluation from simulation-based to physical-based testing. In the physical evaluation during the last phase of the project, digital twins developed in Task 6.4 will be used to bridge the gap between physical and simulation-based testing. The work has focused on the development of a methodology and architecture for guiding this transitioning, from simulation-based testing to lab-based testing, for the SESAME approach and use cases.

This deliverable reports on the work carried out within Task 6.5, describing the final version of the technical components developed for simulation-based testing and how a subset of their output can be assessed in the lab or physical testing. In particular, we present a DSL-supported methodology for guiding this transition, firstly focusing on performing evolutionary simulation-based testing with attention to coverage of the space of potential fuzzing test configurations. A new notion of parameter space structure is presented in order to make this exploration more tractable in the time available for the execution of simulations. Then, an algorithm is presented by which a subset of the test results can be chosen, allowing the users to manage the trade-off between the potentially conflicting objectives of 1) the *exploitation* of the evolved population of the most interesting scenarios in physical testing, 2) the *exploration* of configurations to increase the diversity of selected scenarios and best assess the conformance of simulation to reality throughout the parameter space.

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

1.1 Simulation-Based Testing Challenges

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 [8, 13, 29, 70, 75]. The preference for simulation-based testing is partly due to the high maturity level of modern robotic frameworks (e.g., ROS [59], MOOS-IvP [9] and task-specific custom industrial simulation environments (e.g., Siemens SIMIT, DDD from our partner TTS [71]) that enable the 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 [75]. Also, simulation-based testing supports searching large design spaces of components and system configurations, using limited physical hardware resources, thus reducing the overheads for detecting potential violations of safety requirements significantly [33, 28].

1.2 Physical Testing Challenges

Performing such tests in the real world using the physical MRS can present several challenges. Firstly, testing robotic systems in the real world is time-consuming and expensive [86]. 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 life cycle, which does not give the opportunity to developers to get early feedback on the proposed algorithm and configurations. It is important, therefore, that an available time budget for physical testing is carefully used, in order not to waste the expertise of system testers and that the tests chosen for physical execution are selected judiciously.

Simulated MRS systems can be significantly different in terms of physical features, modelling assumptions and response to transient faults or interference [20, 72]. From the point of view of the testing infrastructure, this *reality gap* has to be taken into account and explained, and if possible, resolved. This reality gap may result from the fact that industrial environments rely on complex topologies and proprietary infrastructure, which presents interfacing challenges. Connecting at runtime to a physical system such as a programmable logic controller (PLC) may require complex interfacing and communication techniques that introduce additional latency and may therefore distort the behaviour of the MRS under test. In addition, simulated components typically use simplified models which diverge from reality in uncertain situations or show transient faults, leading to distinct simulation outcomes at the high level. Reality gaps may also vary under specific environmental conditions such as night-time [61] or under high winds in UAV scenarios.

Therefore, interfacing to real systems with heterogeneity, such as the KIOS and KUKA use cases of the SESAME project can be challenging. 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. Even when common simulation frameworks and standardised robotic middlewares are used, different assumptions may exist in how they are configured for modelling vs simulation which impacts simulation to physical testing accuracy. 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).

In our KUKA use case, we attempted to reduce the interfacing challenges by adding a centralised shared-memory component which acts as a repository of the system state, regardless of whether testing is simulated or physical. Often additional explicit delays or wait states need to be added to allow, e.g. robot positions to synchronise, compensate for jitter in the response times of specific sensors etc. This strategy could be employed, especially in industrial modelling scenarios, to equalise the delays for fuzzing between simulation and reality. For other configurations, we present a method for assessing, exploring and finally, attempting to resolve reality gaps that occur following the completion of physical testing. It may be necessary to tune simulation parameters to produce a match with the observed characteristics of the real world [36, 83] or introduce intentional randomisation into simulated scenarios while testing in simulation in order to close these reality gaps [60, 77].

2 Related Work

2.1 General Simulation-Based Testing

Simulations have been used extensively in the engineering of robotic systems. The proposed work lies at the intersection of model-driven engineering, automated testing, and robotics. This section will present examples, considering general simulation-based testing, distinctions between simulation and reality, and simulation determinism.

Simulation-based testing has been used successfully in the AirSim simulation framework [68], which provides the ability to rapidly investigate a large number of potential configurations for drones and autonomous vehicles. Real robotic system bugs may be detected by simulation-based testing and a majority of real-world bugs may be reproduced for investigation in simulation [75, 70]. 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. [1] propose 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. [34] 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. [67] propose Chrono, a multi-physics simulation package aimed at modelling, simulation, and visualisation of the mechanical parts of ground vehicles. Zhao et al. [85] 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. [76] train models for object localisation on simulated images that transfer to real images by randomising rendering in the simulator. Similarly, Chebotar et al. [17] 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. Dreossi et al present Verifai [26], which proposes a complementary approach to our work for tool-supported testing of cyber-physical systems, typically integrated with AI and machine learning approaches. It uses a monitor to assess the performance of a simulated system at runtime, together with a search process which automatically finds counterexamples and stores them in error tables for later analysis.

2.1.1 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 [18, 19, 65]. 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 [24], BRICS [14], SmartSoft [64], and Robochart [40] 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 [21] and [44].

Despite the available literature on the application of MDE to robotics, the engineering of MRS is still inadequately investigated. Cattivera and Casalaro [15] conducted a systematic mapping study on the application of MDE to the engineering of mobile robots and 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. [27, 48, 69]). Other approaches include Ciccozzi et al. [18], 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 [58] 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. [25] 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].

2.2 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 [84] produces inputs augmented with invalid characters, flipped bits, or including the insertion of known or interesting integer inputs (such as maximum values). TaintScope [81], 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 [41] 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 [39] 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. [23] propose a fuzzer that can be used to identify implementation errors. The proposed fuzzer operates on state machines [12] and generates random values as input keys of state machines. While the two aforementioned fuzzers focus on finding implementation errors, PHYS-FUZZ [82] focuses on finding hazardous scenarios by accounting for physical attributes such as robot dimensions and estimated trajectories. Also, DiscoFuzzer [62] 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 uses models of the system under test, such as state machines and sequence diagrams, to guide the fuzzing process. For example, Schneider et al. [66] proposed a model-based behavioural fuzzing approach where UML sequence diagrams are mutated by fuzzing operators to generate test data. Similarly, SMuF [37] is a fuzzer for Internet-of-Things protocols and generates test inputs that cover 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. Finally, FuzzFactory [45] is a DSL for developing domain-specific fuzzing applications without requiring changes to mutation and search heuristics. This DSL generates the code necessary for a fuzzing application to communicate with a fuzzer and to obtain dynamic domain-specific feedback during the testing process such as performance metrics.

Bohme [11] considers the use of ecological analogies to finding unknown bugs via fuzz testing. Although the number of total bugs is unknown and it is impossible to guarantee that all bugs will be found even in the case in which fuzz testing is run forever, it is possible to use statistical techniques developed by analogy to ecological methods to assess the total number of bugs. The approach presented considers computing discovery probabilities from the smoothed statistics of the previous discovery time, and from the observed number of paths in a program binary discovered so far. Using the hypothesis that almost all information about the number and relative abundance of undiscovered species within the fuzzer's search space is in the number and relative abundance of rare species that have already been discovered, it is possible to estimate the coverage and the number of future defects to be discovered.

2.2.1 Grammar-Based Fuzzing

Grammar-based fuzzing is a technique used to generate better 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 be automatically used to test the compiler [47]. Bonsai fuzzing [79] generates non-trivial tests by using a grammar and an evolutionary process to progressively build up the size of test cases by an evolutionary process. Superion [80] 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.

2.3 Reality Gap and Determinism

It is important to consider the distinction between simulation and reality, since simulation environments may produce dramatically different behaviour than a real environment. For example, in [61], reality gap conditions according to the weather conditions are investigated. The reality gap measured both precision (the number and ratios of correctly matched predictions) and the recall rate with false positives. The scores showed that the reality gap was non-uniform with respect to the testing condition selected; for example, night-time driving produced an increased reality gap. In [20], the authors use time-series analysis to assess the low-level reality gap from the recorded traces of analysis of real controller versus simulated data, using the root mean square (RMS) error from the real behaviour. Then, statistical techniques are used to quantify the reality gap from these time series; e.g., the Pearson correlation coefficient. This approach necessarily provides a lower-level metric for considering the reality gap than considering the deviations in the high-level scenario-specific metrics that we propose in SESAME. However, it could provide a viable debugging technique when investigating the cause of the reality gap.

In a white paper report on reality gap and simulation coverage [2], the matter of reality gap in autonomous driving scenarios is considered. The fusion of novel technology and potentially highly dynamic environments makes such scenarios hard to analyse. The report proposed test selection for coverage be split into *MacroSC* phase; choosing whether certain elements are included in the simulation, and the *MicroSC* phase, which selects particular small-scale interactions. This is similar to our proposed SESAME testing approach in which a DSL is first used to constrain the experimental scenarios performed (leveraging domain knowledge from robotics and systems experts), while the *MicroSC* phase is used for evolutionary optimisation. In testing coverage, a uniformity hypothesis is assumed that equivalent situations are likely to exhibit the same failure behaviour. Since the total space of all possible AV interactions produces an intractable graph, a restricted grid of potential interactions is considered; for example, AV interactions at junctions. This is similar to our SESAME approach in which coverage is determined to be achieved when a certain set of cells in a multi-dimensional space are covered.

In [72], the comparison of self-driving autonomous systems in simulation versus reality is performed, utilising track testing on a small-scale vehicle platform. The results have shown that while model-level metrics (that provide an indication of low-level performance) can frequently generalise, system-level metrics can frequently diverge. For example, the trajectories taken by vehicles can differ substantially. A primary conclusion is that physical testing is still required to validate a scenario, with the proposed hypothesis for most reality gaps being the lack of photorealism (affecting the DNNs) and the inadequate representation of uncertainties in the simulation, including modelling of the throttle and transient spikes in the latency. Therefore, the accuracy of the physical modelling of particular components can be a key factor in simulation inaccuracies.

Two primary approaches for closing the reality gap are system identification and domain randomisation. System identification (SI) consists of improving modelling by the incorporation of identified properties from the world into the simulators. In [16], an example of cable unplugging is used in which the simulator's physical model is tuned to match the real recorded behaviour. The model is shown to be capable of producing low errors when different behaviour. In [36], the authors show that the incorporation of a relatively short period of real-world data (merely a few minutes) can improve performance. SI may also be used in an online manner which allows it to generalise to changing environments [83]. A closely related strategy is model identification strategy: given recorded real trajectories, search for improved simulation parameters to try to reduce the mismatch with the real trajectory [87]. In [38], consideration is given to task-oriented optimisation, which can learn the full parameters such as mass and friction for a dynamics simulation.

Domain randomisation (DR) is an alternative which randomises simulation parameters, typically physical parameters, such as mass and friction, or control parameters, such as time delays and actuator gains. Early work in [35] considered the addition of noise to robotic simulations and showed basic principles; that the addition of moderate noise in evolutionary robotics prevents the evolution of brittle strategies that may miss badly when the environment is changed. In [60], Bayesian methods support domain randomisation with respect to the observed probability distribution. Some strategies consider combinations or more sophisticated applications of randomisation. In DROID [77], a multi-stage combination of SI and DR approaches is used. Firstly human demonstration is used to demonstrate a task. Then, the robot analyses the distributions of parameters from noise and uncertainty in the real world. After this, reinforcement learning is applied to learn a policy for performing the task which is tolerant of the observed uncertainties. This approach can increase the proportion of cases in which the learned policy succeeds at diverse real tasks to 80% from 20% with ordinary DR. In [78], a simple method of random injection is shown to perform relatively well compared to the more complex methods of domain randomisation. Although this random injection is not always the best method, its relative simplicity gives it significant answers in comparison to the more complex methods.

Even with the existence of the reality gap, simulation-based testing is still a viable strategy to explore the design space and expose faults. Sotiropoulos et al. [70] also consider the reproducibility of known extant bugs during fault testing. Their conclusion is that a majority of bugs in robotic software do not require complex physical phenomena to be exposed. Approximately half the theoretically reproducible bugs were actually reproduced in simulation, establishing that even a simulated platform with reality gaps can expose many classes of real bugs. One potential work on considering non-determinism in robotic simulators provides some surprising conclusions. It is frequently assumed that additional repeated executions of a configuration are required in order to discover additional violations. However, the work done in [63] surprisingly determines that the best usage of a limited test budget can be running additional diverse configurations, in order to expose a wider range of potential faults. Although if the purpose is to validate that particular configurations are fault-free, rather than exposing new failure methods (perhaps because these configurations are needed to be safe in real industrial environments) then repeated runs may still be helpful.

2.4 SESAME Testing Process 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 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] and Verifai [26]. There are distinctions in how the fuzzing is specified in SESAME simulation-based testing compared to Verifai, in terms of the flexibility we have in defining activation conditions through condition-based fuzzing, which allows fuzzing to be activated in response to semantic events affecting the simulator.

The aforementioned languages and tools (Section 2.1.1) 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. We explain all these innovations in the following sections.

3 SESAME Multi-Stage Quality Assurance Methodology

3.1 Overall Methodology

The overall methodology for the transition from simulation to physical testing is illustrated in Figure 1, which comprises a number of stages. The human symbol indicates those points which involve the use of human assessment or intervention (robotics and software engineers), and the gear symbol indicates automated processing or code execution. The robotic arm symbol indicates the stages in the methodology which require access to a lab environment and implementation of the robotic scenario in order to execute and assess it (which are also emphasised with a blue background).

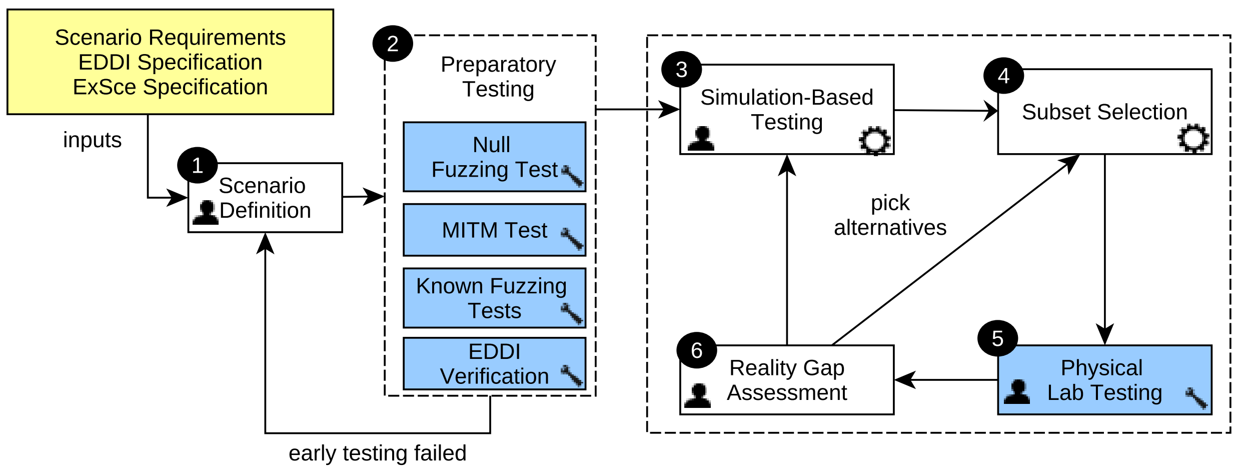


Figure 1: Multi-Stage quality assurance methodology developed for Task 6.5

3.2 MRS Scenario Definition - Step 1

Step 1, which precedes the simulation experiments, begins with the definition of the MRS scenario, its intent and the testing strategy, incorporating an Executable Digital Dependability Identity (EDDI) [53, 54]. This is informed by the inputs indicates to Step 1. These are developed in conjunction with industrial partners, in order to exploit their experience in the intended task of the scenario, the requirements it must fulfil, and the types of failure to which the system could be subjected. The testing criteria as specified in the Executable Scenarios (ExSce) [49] are also considered within this process. This step may be performed multiple times, considering the results of the validation process in Step 2. During Step 1, the following aspects of the scenario and its testing process are specified:

Scenario Requirements: Given the intended mission of the scenario, industrial partners and testing engineers specify the functional and non-functional requirements in order to assess the safety and, potentially, performance of the MRS in achieving the goals of its mission. It is possible for the scenario at this stage to focus on a fragment of the mission, for example, identifying a particular vulnerable component or set of components. This could involve selecting safety violations that could occur and how they can be detected from the available information at simulation time and during execution, and which faults could impact this particular stage of mission execution. Information from fault tree analysis demonstrated in the security-focused parts of the project (e.g., D4.3 [50] and D7.1 [53] could inform this stage of the simulation-based testing, by identifying the types of failures and vulnerable components.

Metrics: Selection of metrics to numerically quantify the defined requirements. For example, the number of violations may be counted during execution, or the maximum intensity or impact of a violation (such as the maximum penetration of a particular safety zone) can be assessed. The choice will depend upon the goals of the industrial user in performing the testing campaigns. Performance characteristics to validate the system's successful completion of its intended goals can also be quantified as metrics; for example, the completion time to survey an expected area using UAVs in KIOS/CCD UAV-based inspection case study.

Fuzzing Operations: These will be defined in consultation with industrial partners and system testers, specifying the types of failures which the industrial partners would like to secure the system. For example, fuzzing operations could model intermittent failure of cabling or wireless signals through transient or permanent message deletion, or components producing values behind schedule by delay operations. Programming errors could be modelled by custom operations. This phase will also consider the maximum intensity of failures to which the system is likely to be subjected, which will inform maximum values of particular parameters (such as message loss rates) that will be used as boundary conditions to constrain fuzz testing. This information may additionally incorporate the experience of the industrial partners and the types of failures they have experienced in deployment.

Timing and activation conditions: In this stage, consideration is given to when fuzzing is applied, taking into account the phases of the mission and the activities particular robots will be performing as it continues. This may manifest as specifying time ranges in which each operation could be activated, as boundary conditions for the search space. Alternatively, in more dynamic missions, conditions based upon current system status may be used to trigger and end the fuzzing. In this case, consideration must be given to how the necessary state information to evaluate these is obtained.

EDDI Specification The MRS system may incorporate an EDDI to provide dependability guarantees in the presence of failures. In this stage, it is important to both specify the EDDI and determine how it will interact with the fuzzing experiments. The EDDI may be used as a monitor to detect/avoid pending violations, and the testing campaign would amount to verifying how it behaves in the presence of fuzzing. Defining an interface for the EDDI requires specifying the information that the EDDI requires from the system, the actions that it performs, and how this interacts with the testing campaign. Different possible EDDI instantiations exist; an example using conditional safety certificates (ConSerts) is presented in Section 6.1.7.

Simulation augmentations: It may be the case that data required for either simulation performance monitoring with the given metrics, or the information required as input to the EDDI is not directly available from physical sensors present in the system. In this case, it will need to be computed within the simulator and fed to the testing platform. For example, testing for collision violations, or entry to a forbidden safety zone, would require additional logic beyond merely the robotic joint input values - such as 3D intersection checking. In this case, code would have to be implemented within the simulation that computes the position of the robot and determines collisions, and during physical testing, an instance of the simulator would need to be executed, informed by data from the physical system. Therefore, these simulated computations may itself be subject to reality gaps, which can be assessed in Step 2.

3.3 Preparatory Testing - Step 2

The purpose of preparatory testing is to validate the scenario as defined in Step 1 in the lab environment. The first required aim is to ensure conformance between the simulation and the physical system in a test case with no fuzzing. Reality gaps between simulation and physical systems, even in the absence of fuzzing could result from a variety of sources: different delays in communication with the physical system and in simulation; simulation algorithms with models that do not precisely match with the physical system; limited or incomplete computation of augmented values in simulated models fed by physical system information; and noise/transient signals that trigger violations in a real system that would not be present in the simplified environment in which all components behave according to simplified models. If any of the stages in Step 2 fail, then it may be necessary to return to Step 1 to attempt to identify the cause and potentially alter the scenario to be more robust to the identified issues affecting determinism. If knowledge or expertise about some of these steps is already available and sufficient evidence exists about the reliability and conformance, then some of these steps may be ignored.

3.3.1 Null Fuzzing Validation - Step 2A

This stage consists of executing the scenario in the lab environment and monitoring for violations of the defined safety requirements, or failure to fulfil the necessary performance requirements. If violations of the safety requirements occur in the fuzzing-free simulation, or in the physical system; it is important first to check the scenario and its monitoring infrastructure, i.e., performance metrics are correctly implemented. A useful approach here is to check the low-level accuracy of input time series that are used to determine the metric values and compare them against the simulated scenario. This will allow, for example, verifying the correct position of the particular robots, or inconsistent placement of collision detection zones. Recording this information will allow to determine whether there may have been noise or another transient value that triggered momentarily a violation in the real system, or the tolerances of the safety zones which have been defined could be insufficient.

3.3.2 MITM Testing - Step 2B

During physical system testing, the next step in the testing process is to forward simulation variables through the man-in-the-middle (MITM) mechanism used in the SESAME simulation-based testing platform. This involves receiving incoming variables from the simulator via a custom simulation-specific mechanism, and then re-transmitting them back to the simulator with potential modifications. This MITM mechanism is described in more detail in Section 3.4. Even when no explicit fuzzing operations are applied within the platform, the delays which are introduced through this mechanism could impact the overall system performance and stability, especially when fuzzing high-frequency simulator variables that form part of a control loop. Therefore, it is important to thoroughly determine the baseline impact that the introduction of this MITM will have upon the system. This should be done in stages, incrementally increasing the number of simulator variables forwarded via the platform. The consequences of this should constantly be assessed by examining the system behaviour for stability, metric violations, and overall determinism upon repeated runs. This will allow any impact on the physical system from the interconnection with the testing platform (potentially from any increasing latency in processing and handling messages), to be isolated and quantified. This may involve specific tests to measure the baseline latency in this case.

3.3.3 Known Fuzzing Testing - Step 2C

Following MITM testing, a series of known fuzzing tests can be performed, to ensure physical system stability and response under certain known fuzzing configurations. These tests have been manually defined in order to activate fuzzing at specific times, and to produce a known response in terms of the given scenario metrics. The first purpose of known fuzzing testing is first, to ensure that the system remains stable under fuzzing, with no unexpected or potentially dangerous behaviour exhibited as a result. Also, verifying the outcome of the known fuzzing tests to ensure that they produce the same effect in the physical system as observed under simulation. If there is a significant reality gap during this process, it may be necessary to return to the physical scenario design phase in order to alter the intended performance metrics. Changes may include, for example, switching to alternate metrics to measure the same requirement, for example, in KUKA's use case measuring the depth of penetration of a safety zone, instead of the number of incidents of penetration.

3.3.4 EDDI Verification - Step 2D

At this stage, it may also be useful to verify the EDDI on data recorded from the real system. The purpose of this EDDI verification is to determine in test cases whether the EDDI takes the same decisions that it would do in the physical system as under simulation, and if discrepancies occur, to expose their causes. It may be possible, for example, for the transition to physical testing to add delay variance, altering the relative timing of signals to which the EDDI subscribes. This may impact the EDDI's decisions by altering is behaviour when the input preconditions of one of its checked conditions are not simultaneously true. Evaluating the EDDI's response on a set of known fuzzing tests could help to expose these situations in advance.

3.3.5 Transition from Simulation to Physical Testing

The right-hand box of Figure 1 defines the iterative process which will be followed in order to transition from simulation-based to lab testing. This process involves first performing simulation based testing, and then selecting a subset of its output for physical lab testing. Depending on the output of the process and its resulting reality gaps, this process may be altered and repeated as necessary. In Section 4, we describe in detail the simulation-to-physical testing transition and in the following section, we describe the simulation-based testing process in Step 3.

3.4 Simulation-Based Testing Process - Step 3

This section provides a summary of our SESAME simulation-based testing process, considering the application of model-driven engineering (MDE) and evolutionary optimisation for system testing. A fully-fledged description is presented in D6.2 [52]; this section will summarise it as well as highlight changes and further developments since that deliverable.

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 and the employed EDDIs. Identifying these edge scenarios using the simulation-based testing framework entails following the simulation-testing process presented in Figure 2. The four steps of this process (3.1-3.4) are contained under Step 3 of the larger integrated testing process (cf. Figure 1). The steps 3.1-3.4 are presented below:

Simulation-Testing - Step 3.1 The users specify the testing space specification and MRS structure model. We provide the specification of the testing DSL in Section 3.7. The MRS structure model corresponds to the 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. The MRS structure model provides an encoding of the earlier scenario setup step (Step 1) in the wider integrated methodology of Figure 1.

Simulation-Testing - Step 3.2 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.

Simulation-Testing - Step 3.3 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

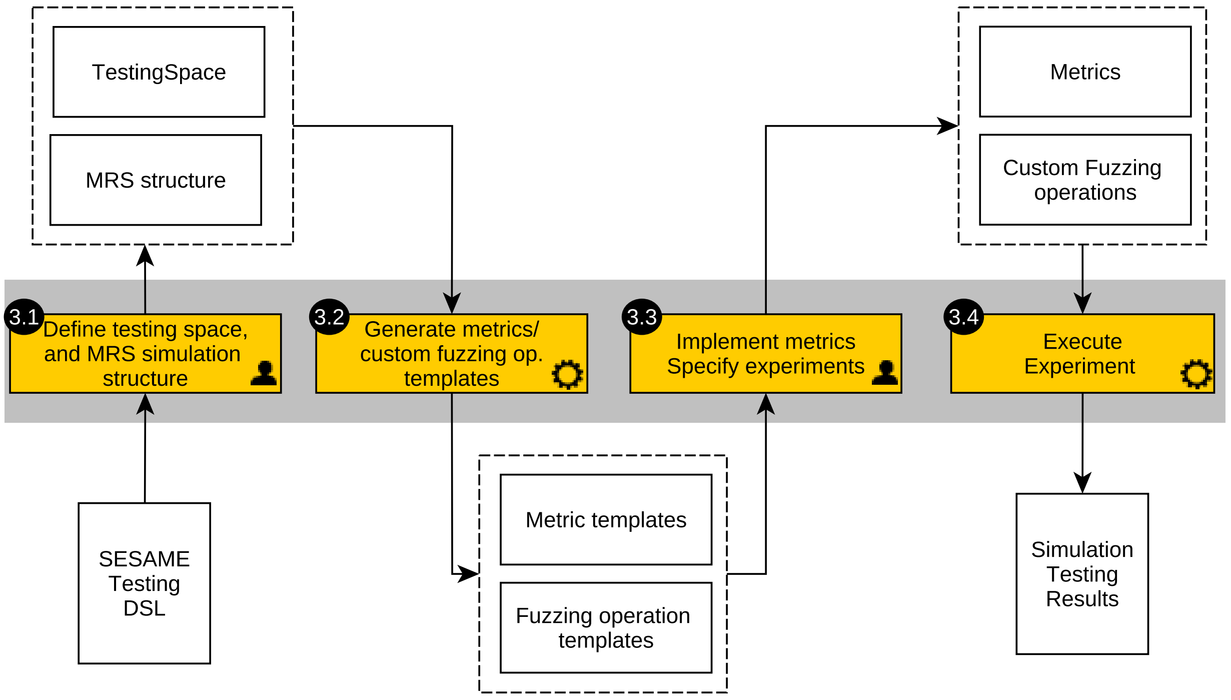

TestingSpace
MRS structure
Metrics
Custom Fuzzing operations
3.1 Define testing space, and MRS simulation structure
3.2 Generate metrics/ custom fuzzing op. templates
3.3 Implement metrics Specify experiments
3.4 Execute Experiment
SESAME Testing DSL
Metric templates
Fuzzing operation templates
Simulation Testing Results

Figure 2: Simulation-based testing methodology

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 should be included.

Simulation-Testing - Step 3.4 The SESAME simulation-based testing platform experiment runner then performs a given experiment. The experiment runner dynamically generates and launches repeated iterations of tests according to a given strategy, e.g., evolutionary, repeated execution to verify performance. 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 evaluates each test by first dynamically generating a specialised test runner which acts as a middleware, communicating with the low-level simulator via a simulator-specific interface, and using any custom-supplied metric definitions provided in Step 3.2 to quantify the impact of the fuzzing test. This information is communicated to the experiment runner and may be used to guide a multi-objective optimisation process.

3.5 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 3.7, that specifies the fuzzing space, available fuzzing operations, and the range of their parameters for experiments. The DSL is paired with a model-driven code generation engine that consumes DSL-compliant models and generates a middleware component to interface with the target simulator.

The middleware fulfils a twofold role. First, it mediates between the fuzzing engine and simulator at runtime, passing messages back and forward to allow message modification during the execution of a selected fuzzing test. Secondly, the middleware enables the monitoring and recording of communication of specific message

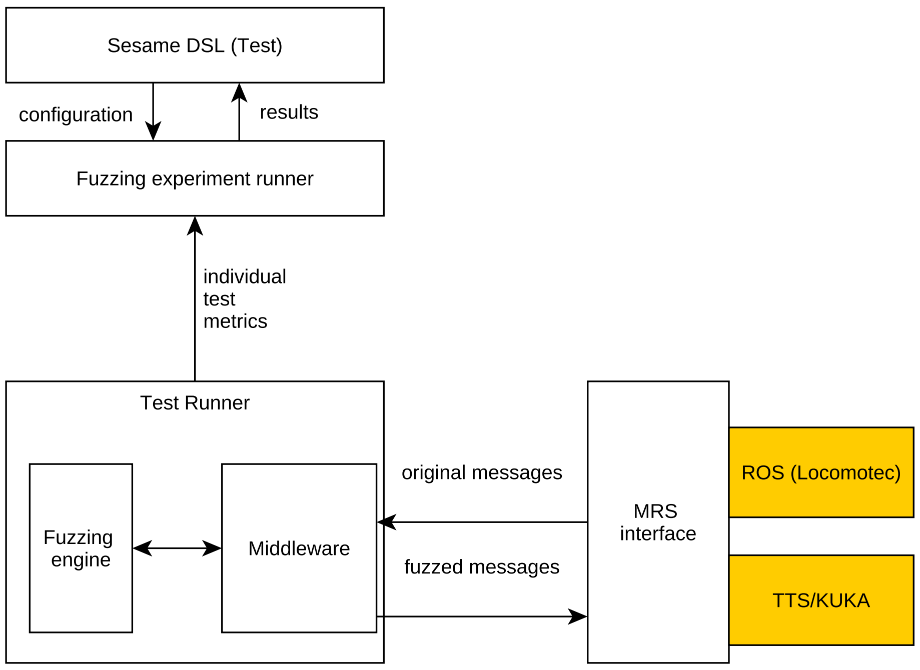


Figure 3: The architecture of the test runner and its runtime connection to the MRS

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. On the robotic simulator side, a 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.

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' (MITM) 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.

3.6 Fuzz Testing Dimensions

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 (e.g., 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 3.8.

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 a 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. The SESAME platform supports both condition-based fuzzing in which both the start and end of fuzzing can be in response to scenario-specific conditions, and condition-based time limited (during which the fuzing event start is in response to a specific condition, but the time length of fuzzing is fixed).

3.7 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 increases the abstraction level and provides configurability by system testers as well as allowing results to be exported to other tools. In addition, our testing platform provides a convenient visual editor for system test engineers to configure the MRS during 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 TestingSpace class is the root element of the DSL, and contains permissible fuzzing operations (under *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 regard 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 selection allows the user to specify the parameters for an experiment by selecting one of several subclasses. For example, including NSGAEvolutionaryAlgorithm allows an evolutionary experiment with the NSGA-II algorithm [22], and contains specific parameters relevant to this approach, e.g., the number of iterations and the population size. In this case, the evolutionary algorithm is concerned with maximising violations of the intended result metrics, and does not specifically track the coverage obtained. We also provide a new coverage-aware GA NSGACoverageWithCells, which seeks to improve coverage of the space of potential fuzzing tests. Further, RepeatedRunner provides support for repeated execution of a particular selected test a number of times. The utility of this is to allow an interesting test with a high reality gap or other performance issues to be repeated and the reasons for its behaviour investigated in depth. Regardless of the test generation strategy selected, 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 and finalised upon their completion, containing references to the population of results upon a Pareto front. This is an important feature that enables keeping track of the history of evolved tests during simulation-based testing.

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. 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 a 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 3.8 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 the 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.

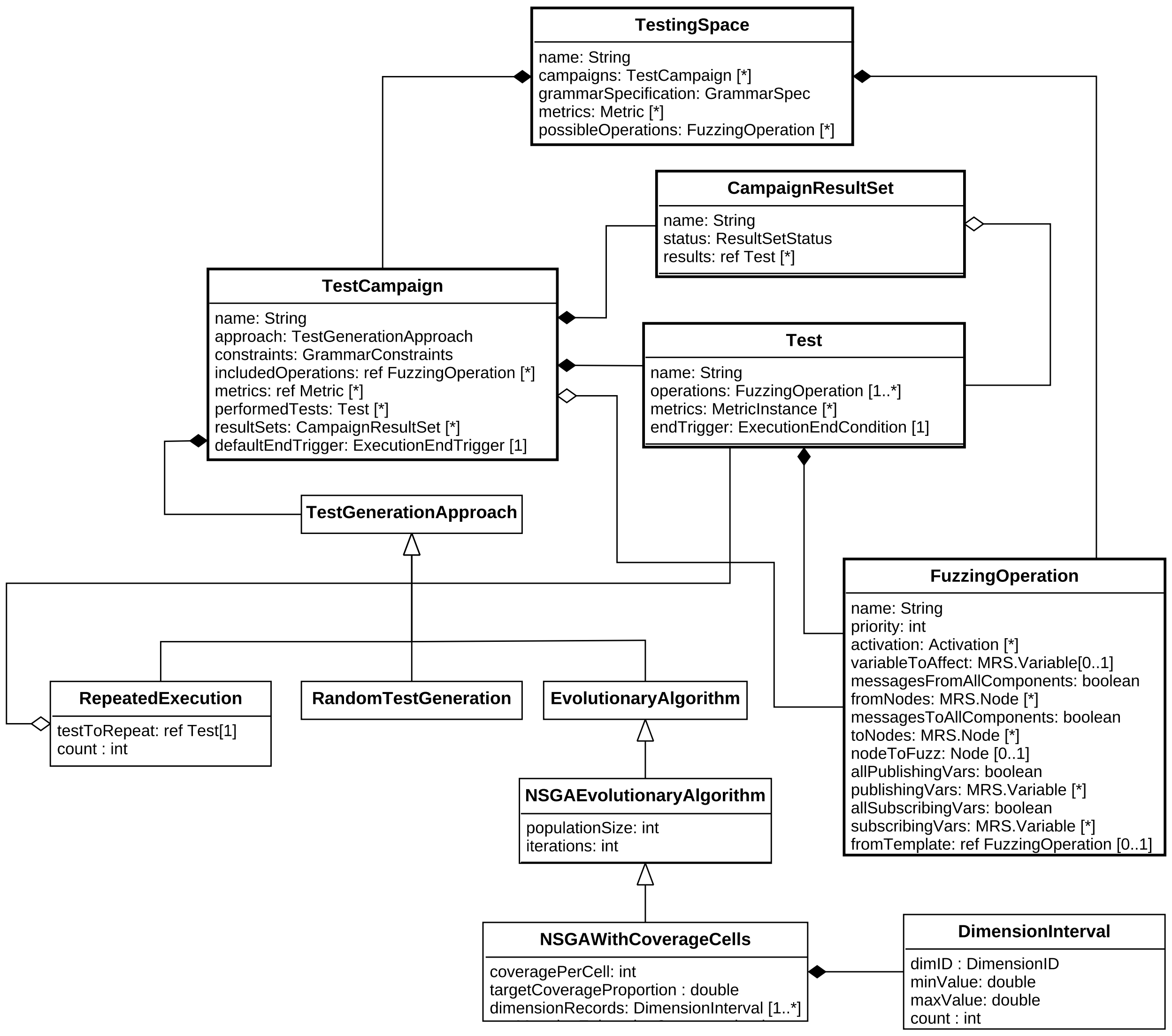

TestingSpace
name: String
campaigns: TestCampaign [*]
grammarSpecification: GrammarSpec
metrics: Metric [*]
possibleOperations: FuzzingOperation [*]
CampaignResultSet
name: String
status: ResultSetStatus
results: ref Test [*]
TestCampaign
name: String
approach: TestGenerationApproach
constraints: GrammarConstraints
includedOperations: ref FuzzingOperation [*]
metrics: ref Metric [*]
performedTests: Test [*]
resultSets: CampaignResultSet [*]
defaultEndTrigger: ExecutionEndTrigger [1]
Test
name: String
operations: FuzzingOperation [1..*]
metrics: MetricInstance [*]
endTrigger: ExecutionEndCondition [1]
TestGenerationApproach
FuzzingOperation
name: String
priority: int
activation: Activation [*]
variableToAffect: MRS.Variable[0..1]
messagesFromAllComponents: boolean
fromNodes: MRS.Node [*]
messagesToAllComponents: boolean
toNodes: MRS.Node [*]
nodeToFuzz: Node [0..1]
allPublishingVars: boolean
publishingVars: MRS.Variable [*]
allSubscribingVars: boolean
subscribingVars: MRS.Variable [*]
fromTemplate: ref FuzzingOperation [0..1]
RepeatedExecution
testToRepeat: ref Test[1]
count : int
RandomTestGeneration
EvolutionaryAlgorithm
NSGAEvolutionaryAlgorithm
populationSize: int
iterations: int
NSGAWithCoverageCells
coveragePerCell: int
targetCoverageProportion : double
dimensionRecords: DimensionInterval [1..*]
DimensionInterval
dimID : DimensionID
minValue: double
maxValue: double
count : int

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

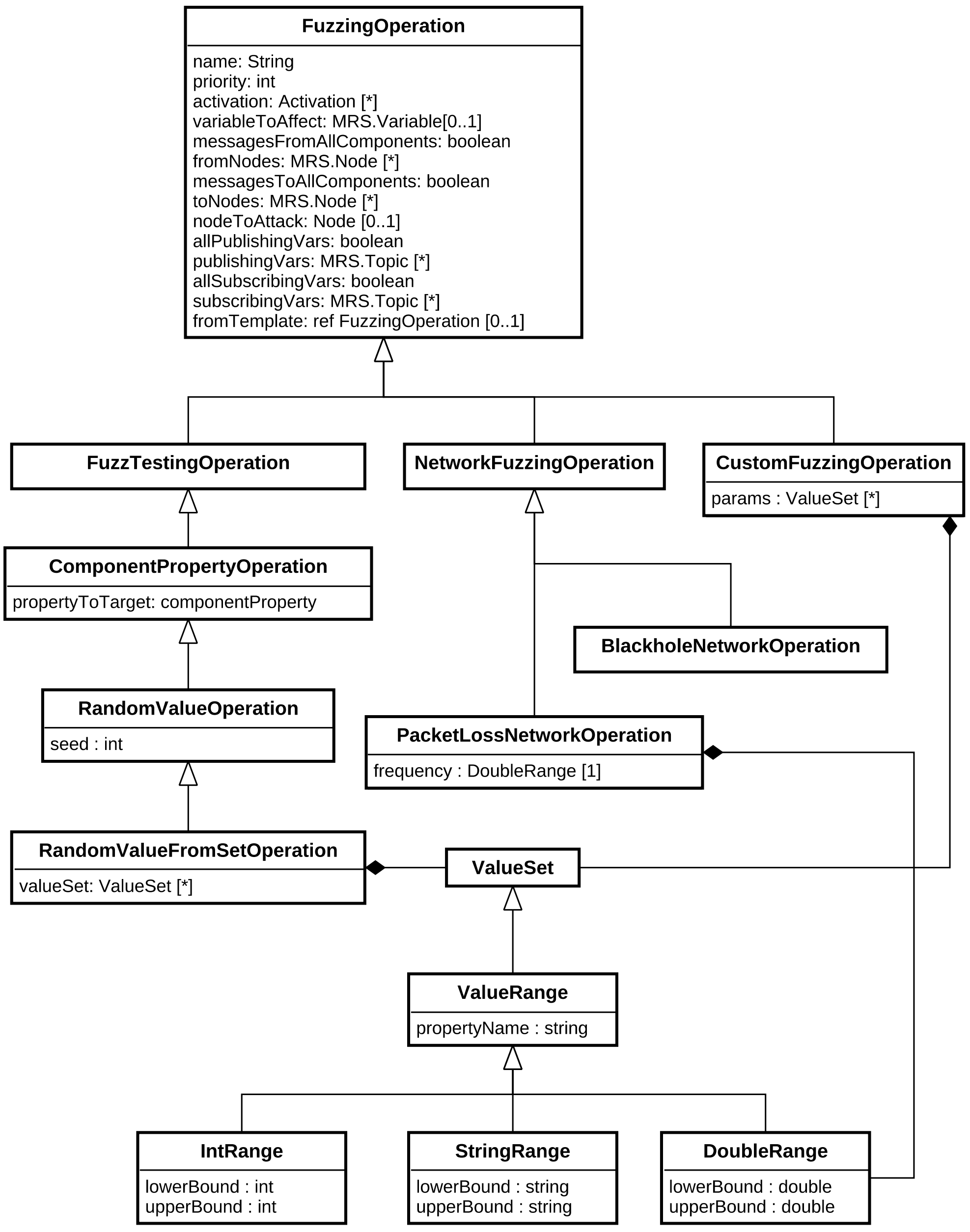

FuzzingOperation
name: String
priority: int
activation: Activation [*]
variableToAffect: MRS.Variable[0..1]
messagesFromAllComponents: boolean
fromNodes: MRS.Node [*]
messagesToAllComponents: boolean
toNodes: MRS.Node [*]
nodeToAttack: Node [0..1]
allPublishingVars: boolean
publishingVars: MRS.Topic [*]
allSubscribingVars: boolean
subscribingVars: MRS.Topic [*]
fromTemplate: ref FuzzingOperation [0..1]
FuzzTestingOperation
NetworkFuzzingOperation
CustomFuzzingOperation
params : ValueSet [*]
ComponentPropertyOperation
propertyToTarget: componentProperty
BlackholeNetworkOperation
RandomValueOperation
seed : int
PacketLossNetworkOperation
frequency : DoubleRange [1]
RandomValueFromSetOperation
valueSet: ValueSet [*]
ValueSet
ValueRange
propertyName : string
IntRange
lowerBound : int
upperBound : int
StringRange
lowerBound : string
upperBound : string
DoubleRange
lowerBound : double
upperBound : double

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

3.8 Fuzzing Operations Specified within the DSL

This section 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 fuzzing operations (presented in Figure 5) are available:

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: 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.

In order to provide scenario-specific functionality, custom fuzzing operations can also be employed in order to provide additional operations. These operations could, for example, alter calibration or other configuration file entries during a pre-processing phase, as well as perform arbitrary message transformations at runtime. The class **CustomFuzzingOperation** provides this functionality. The name of the custom operation specifies a Java class which users implement to define how the operation processes incoming MRS messages. The CustomFuzzingOperation contains a set of user-specific parameters; subclasses of ValueSet. These parameters define the state for this specific operation; for example, if using a fuzzing operation to amend a custom data structure, a StringSets can quantify which point, or its filesystem paths. PointRange can be used to define a range of points to be used as an offset, which will be reduced to specify specific values during the evolutionary process.

3.9 Fuzzing Modalities

3.9.1 Time-Based Fuzzing

Using as input the fuzzing specification model that defines the testing space (and conforms to the metamodel from Figure 4), the time-based fuzzing method evolves a set of testing campaigns aiming at establishing the critical intervals (in the temporal dimension) during which fuzzing can trigger safety requirements violations. To achieve this, the time-based fuzzing modality employs the simulator's timing reference and statically defined start and end times as specified in the evolved fuzzing test. When the start time specified within the fuzzing campaign is reached, the fuzzing operation becomes active and is applied to incoming messages upon that variable. Once the end time is reached, the fuzzing operation becomes inactive again, and messages will be forwarded to their recipient components unchanged.

3.9.2 Condition-Based Fuzzing

Given the fuzzing specification model, constructed based on the metamodel in Section 3.7, condition-based fuzzing aims at discovering critical events that are considered main contributors to deviations in safety requirements. A condition corresponds to a state of the robotic system during mission execution governed by logical information derived from the mission and system components, which supports the activation of fuzzing based

Table 1: Grammar in EBNF for condition-based fuzzing

⟨compCond⟩	::=	⟨compCond⟩⟨logicOp⟩⟨compCond⟩ |⟨basicCond⟩
⟨basicCond⟩	::=	⟨expr⟩⟨binComp⟩⟨var⟩
⟨expr⟩	::=	⟨var⟩ | ⟨int⟩
⟨logicOp⟩	::=	*AND* | *OR*
⟨binComp⟩	::=	*lessThan* | *greaterThan* | *equal*
⟨int⟩	::=	*1* | *2* | *3* | ...
⟨var⟩	::=	$\mathit{ConditionVariable}_1$ | ... | $\mathit{ConditionVariable}_n$

on the specification of complex spatial and geometric properties. Its evaluation requires using simulator variables such as the robot's current location to establish if the distance clause is satisfied. Evaluating a condition therefore determines at a particular instant whether it holds or not. For instance, in a UAV inspection use case, a condition could be "the UAV's distance to an inspection surface is less than 5m". Although the condition-based fuzzing approach was proposed as a potential approach in our earlier deliverable, its realisation and implementation is a novel contribution in the final version.

The premise underpinning the condition-based fuzzing instance is the evolution of semantically meaningful testing campaigns. Conditions are used to trigger the activation and deactivation of fuzzing when a given event expressed in boolean logic occurs. These events are built from expressions using scenario-specific condition variables, whose values can be provided from our metric infrastructure. Our hypothesis is that conditions are more informative and explainable, and require less introspection in order to understand the overall effect of the testing campaign. 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).

Conventional condition-based fuzzing is controlled by start and end conditions. For each fuzzing record in the inactive state, the start conditions will be evaluated at runtime on every simulation event in order to determine a boolean status. If the condition evaluates to true, that fuzzing record will be set to active. Then, for every applicable event, the fuzzing operation will be invoked to transform the incoming events for that fuzzing record. Once activated, the fuzzing record remains active until the end condition evaluates to true. Once the end condition is satisfied, fuzzing is deactivated for this operation. We assume that conditions can only be activated once, such that a record which is activated is no longer available to be reactivated later. Relaxing this assumption to support repeated activations is definitely feasible and requires modest engineering effort.

An alternative type of condition-based fuzzing consists of a start condition with a timing length. Fuzzing will be activated upon the start condition evaluating to true, and will be active for the given length of time. Following the expiry of the statically defined time, the condition will be inactivated. The purpose of condition-based time-limited fuzzing is to support situations in which fixed bursts of fuzzing have to be applied in response to a semantic condition, and also to assist in debugging particular situations in which the fuzzing operation may interfere with determinism by itself affecting the timing of execution of the end condition.

Another key concept of condition-based fuzzing is to encompass the possible space of conditions within a grammar, constraining the synthesis of conditions that activate and deactivate the fuzzing records within a testing campaign [10]. Table 1 shows the generic condition-based fuzzing grammar in the Extended Backus-Naur Form (EBNF) [46]. This grammar can be auto-generated from the DSL information incorporating the permitted condition variables in a given testing campaign. 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). The production rule for the non-terminal symbol *var* expands into the set of grammar variables specified in the model instance that obtain their values from performance metrics specified in Figure 4. Since these grammar variables are mission-specific, their definition requires input from domain experts. At the same time, however, these variables are the only mission-specific part of the grammar. This important characteristic enhances the generality of the approach, enabling its application to other domains and missions without altering its core production rules.

4 Transitioning From Simulation To Physical Testing

4.1 Introduction

In this section, we consider the transition which will occur from simulation-based testing to the physical testing in a lab or real-world environment. This process will involve multiple considerations which are listed below. In each consideration their relationships to the stages of the methodology in Figure 1 are presented:

Efficiently selecting a subset of configurations for physical execution (Step 4)
Because the simulation-based testing process is much faster and more economical to execute than physical testing [33, 28], a larger and more diverse set of configurations can be explored in simulation-based testing than could be tested physically. Therefore, the physical testing scenarios chosen for execution should be selected in order to carefully utilise the limited physical testing time and resources, while providing information on both the conformance between simulation and reality, together with validation of the real testing results.

Ensuring a safe mapping from simulation to physical testing (Steps 4 and 5)
Simulation-based testing can easily and safely explore configurations leading to physical impact, collisions or other undesirable situations which present risks to the safety of human testers or others, or physical damage to robotic or other hardware. These potentially destructive settings may not be executed physically without unacceptable costs or risks. Therefore, it is necessary for the selection of physical testing configurations to ensure that chosen scenarios can be physically tested in a safe manner. One method to achieve this is scenario-specific substitutions that involve backup safety hardware. For example, in a scenario in which the physical hardware provides emergency safety interlocks (e.g. KUKA use case), these can be used to prevent motion that would lead to a collision in a real fuzzing scenario, and activation of these interlocks may be considered equivalent to a safety violation under simulation. Safety configurations for physical testing are thoroughly considered in Section 4.3.

Quantifying reality gaps (Step 6)
Simulation necessarily provides a simplified model of the real system, which means that its performance could deviate from reality in important instances. Therefore, it is important to assess the quality of the simulator as a predictor of real system performance, and to explore the causes of any discrepancies. When simulation results diverge from reality, the simulation may predict failures that do not occur in the real system (false positive) or more dangerously, could fail to predict failures that occur in real testing. In the simplest case, reality gaps could be systematic, occurring under any usage of a fuzzing operation. However, reality gaps may also be localised (only occurring under specific fuzz testing configurations exhibiting combinations of operators, timing and other parameters). Therefore, physical testing must be diverse and examine a sufficient number of configurations to have confidence that the simulation results are a good predictor of real behaviour.

Potential compensation for reality gaps (Step 6) In the cases in which a reality gap exists, it is important to consider its source and how it can be handled. If the reality gap occurs as a result of inadequate modelling of a particular component, this model can potentially be improved and the simulation re-executed. Another potential strategy is to use domain-based randomisation, in which some noise obtained from the physical environment is added to the real system at simulation time in order to provide a better match between simulation and reality. This will be explored further in Section 4.7.

4.2 Simulation-Based Testing As Part of The Integrated Methodology

In the previous section, we considered the implementation of simulation-based testing. Now, we consider how simulation-based testing fits into the integrated testing methodology. The purpose of the simulation-based testing phase is to exploit the advantages of robotic simulation (fast execution, and no prospect of damage to hardware) in order to widely explore the space of potential fuzzing configurations and find those with the greatest impact. Here, we consider how modifications to the GA and its search process can quantify the coverage of the space of potential fuzzing operations, which is useful for more diverse physical testing.

The space created by variations of applied fuzzing will be referred to as the *parameter space*, while the output results which are obtained during the simulation (metric values) are referred to as the *result space*. There are a number of aspects of variability to a fuzzing configuration that can be considered in terms of defining the parameter space. These were fully described in Section 3.6; here, we summarise the aspects considered in the transition to physical testing.

Fuzzing operation choices: The chosen fuzzing operation; for example, delay, deletion or value modification

Fuzzing parameters: The intensity of fuzzing that is applied; for example, in the case of message delay, the boundary to the time length of particular delay applied

Fuzzing activation: When fuzzing is activated during simulation - either at statically specified start-end time regions or in response to particular *conditions*. Conditions used in condition-based fuzzing are described in Section 3.9.2.

The unrestricted *parameter space* of potential fuzzing tests is likely to be too large to be efficiently searched in simulation testing experiments, especially as fuzz testing scales up to larger numbers of simulator variables and available operation parameter set choices. This occurs since the size of the fuzzing configuration space scales exponentially with the number of potential fuzzing operations included in the 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 a minimal bound to the number of potential configurations in the fuzzing search space is at least $(OP)^V$. The fuzzing configuration space is in fact significantly larger than this, due to continuous choices in the timing in which fuzzing operations are active, the possibility of multiple repetitions of operations, and variations of parameters. Although one strategy of our simulation-based testing is to partition the space into a distinct series of experimental campaigns (e.g., by selecting particular robots or combinations of robots to focus upon), every fuzzing configuration requires evaluation in simulation which will take likely several minutes for high fidelity simulation. Therefore, it would not be tractable to provide full coverage of this search space. A full treatment of the issues involved in coverage is given in our previous deliverable D6.2 [51].

Therefore, this section presents a strategy to reduce the size of this full fuzzing configuration space to create a smaller parameter space, of which coverage of a reasonable proportion can be reasonably achieved. Part of this approach is to use the expertise of system testers to constrain the space, for instance, by testing early cases with a restricted number of robots selected by these engineers as most likely to produce violations (reducing the number of affected simulator variables, and therefore a reduction in V). The scenario definition phase of our system testing methodology in Step 1 of Figure 1 allows this reduction by specifying the boundaries of viable fuzzing operations, activation timing and parameters ranges for the entire space in the DSL for particular experiments. 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.

4.2.1 Dimensionality Reduction Of Parameter Space

In Section 4.2, the importance of reducing the size of the fuzzing configuration parameter space to a more manageable size was discussed. This *parameter space* can be conceptualised as a set of dimensions that represent numerically the variability of fuzzing aspects within an experiment. The intent is that adjacent or closely spaced points within this space share some sort of commonality with respect to one or more aspects of fuzzing. A single fuzzing test configuration (consisting of a number of fuzzing operations) is represented by a single point within that space. Although this necessarily loses some of the complexity in regards to the semantics of the fuzzing involved, this reduced space is more practical for coverage during the simulation-

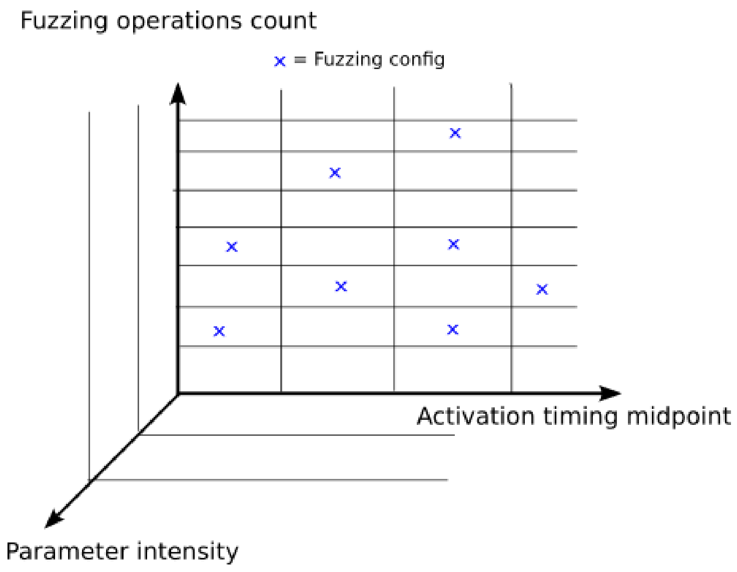


Figure 6: An example 3D Parameter space

based testing experiments. It is important that simulation results cover this reduced parameter space, in order to sample widely the potential dimensions of variation.

An example of a simple three-dimensional parameter space is presented in Figure 6. Three-dimensional grid cells indicate the divisions of the parameter space into specific segmented regions. In this diagram, the x-axis (horizontal) represents the midpoint of the timing activation point of fuzzing under a particular configuration, the y-axis (vertical) the number of fuzzing operations in the configuration, and the z-axis (diagonal - into or out of the page plane) aggregate information about the fuzzing parameter intensity. Every blue cross in the diagram indicates a tested fuzzing configuration explored during an evolutionary fuzzing campaign.

In practical use cases, the dimensionality of the parameter space will actually be considerably higher than three; since each aspect of fuzzing variation could be described by multiple properties. The mapping described below, referred to as the *SESAME Standard Parameter Reduction - SSPR* specifies our proposed strategy for dimensionality reduction, used in this report and proposed for physical experimental evaluation.

Fuzzing Activation Information. The first aspect of variability with regard to fuzzing activation concerns the timing of the fuzzing operations; when fuzzing activation is active during the execution of the selected scenario. In Section 4.2, three modes of fuzzing activation were discussed: time-based and condition-based, and condition-based time-limited fuzzing. Depending on the mode of fuzzing activation used in experiments, the activation timing may be known in advance, or only known following simulation execution. More specifically, in time-based fuzzing, the fuzzing configuration gives the set of start and end times of each fuzzing operation directly. When using a form of condition-based fuzzing, the scenario must be executed in order to determine the observed timing, recording the time at which fuzzing is activated when the starting condition is first activated. Even when fuzzing end times are defined precisely, as in the condition-based time-limited case, it is important to recover the actual time.

For all forms of fuzzing, therefore, information is available in the model after test completion that defines the start and end times of fuzzing for every operation. Since this information is computed on the fly, we can proceed with the dimensionality reduction. The proposed SSPR approach reduces the fuzzing activation time interval sets to three independent timing dimensions T_1, T_2 and T_3; each captures a different statistical aspect of timing variation. The values for the timing dimensions $T1, T2, T3$ for a fuzzing configuration can be computed as follows:

- For all fuzzing operations times defined in a test configuration:
- Compute its midpoint as the mean of start and end timing
- $T1 \leftarrow$ the mean of fuzzing interval midpoint

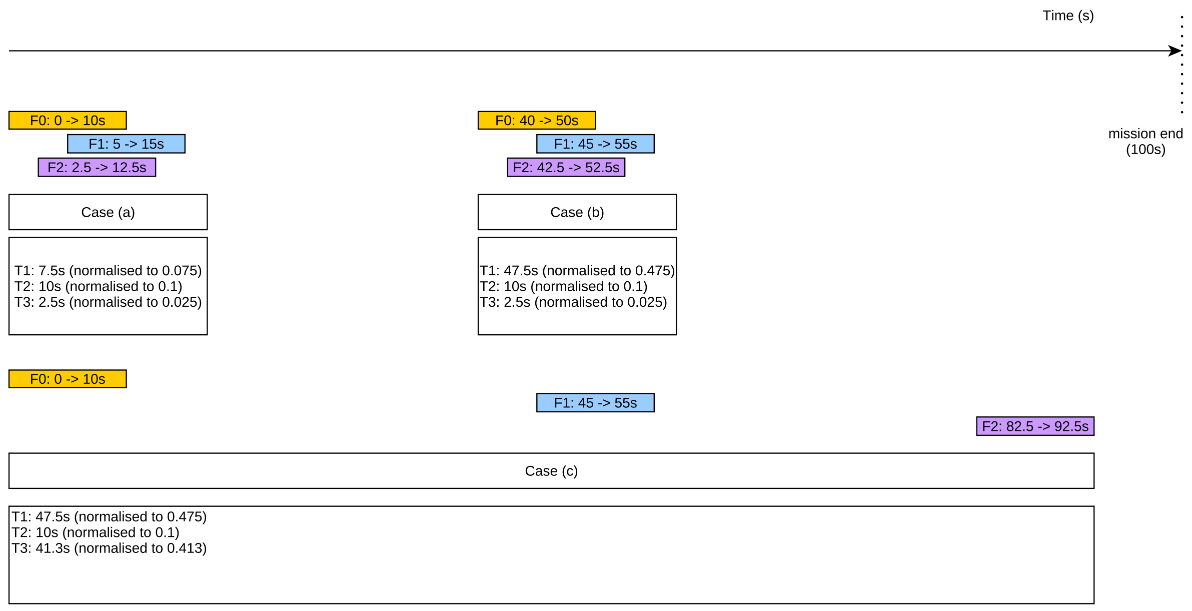

Time (s)
mission end (100s)
F0: 0 -> 10s
F1: 5 -> 15s
F2: 2.5 -> 12.5s
Case (a)
T1: 7.5s (normalised to 0.075)
T2: 10s (normalised to 0.1)
T3: 2.5s (normalised to 0.025)
F0: 40 -> 50s
F1: 45 -> 55s
F2: 42.5 -> 52.5s
Case (b)
T1: 47.5s (normalised to 0.475)
T2: 10s (normalised to 0.1)
T3: 2.5s (normalised to 0.025)
F0: 0 -> 10s
F1: 45 -> 55s
F2: 82.5 -> 92.5s
Case (c)
T1: 47.5s (normalised to 0.475)
T2: 10s (normalised to 0.1)
T3: 41.3s (normalised to 0.413)

Figure 7: Example dimensional reduction for dimensions T1-T3 in fuzzing operation timeline

- $T2 \leftarrow$ the fuzzing interval mean length
- $T3 \leftarrow$ the fuzzing interval midpoint standard deviation

An example of this dimensional reduction process is shown in Figure 7. In this example, three different potential fuzzing configurations (a), (b) and (c) are shown on a timeline from 0 to 100 seconds, together with their computed dimensional values for $T1 - T3$. Each configuration contains three fuzzing operations F0, F1 and F2. In the first two cases (a) and (b), all three fuzzing operations are closely aligned temporally, with (a) close to the simulation start and (b) around the middle. However, in the third case (c) fuzzing operations are distributed widely throughout the experiment, with F0 at the start, F1 in the middle, and F2 at the end. It can be seen that in this case, the mean value T1 is the same for cases (b) and (c). But the standard deviation computed in T3 is what separates these configurations, in terms of showing the temporal clustering of the operations throughout the simulation.

Fuzzing Parameter Information. Every fuzzing operation in a fuzzing test may have parameters which control its intensity or impact on the system, with semantics specific to that particular operation. For example, message delay operations include a parameter which defines the maximum or minimum delay injected, while value alteration could include the range of permitted alterations to be performed on the chosen message field. It is important to normalise all these parameter values by dividing them by a permitted range, in order to ensure that the values are comparable. Although the precise semantics of the operations are different, this will provide a general consideration of the intensity of particular operations.

In all fuzzing operations, the parameters are defined ahead of time before the execution of the test. Similar to the case of the activation timing, the SSPR approach reduces the fuzzing parameter information for all operations given to two dimensions P1 and P2:

- For all parameters of all fuzzing operations:
- If the fuzzing parameter is a single numerical value, set P_V to its value directly
- If it is a FuzzRangeParameter, set P_V as the width of the range of the FuzzRangeParameter (upper minus lower)
- Obtain $\hat{P}_V$ (normalised P_V) by dividing by the validity range. This can be found as the maximum and minimum range allowed for the fuzzing configuration in the DSL

- Set P1 to the mean of all normalised fuzzing parameter $\hat{P}_V$ in this test
- Set P2 to the standard deviation of all normalised fuzzing parameter $\hat{P}_V$ in this test

Fuzzing Operation Information. When considering fuzzing operations allocations, there are two factors involved; the fuzzing operation selected, and the variable chosen. In this situation, we consider the fuzzing operation selection, since we rely upon the experimental scenario selection (for the selection of the chosen campaign) to constrain the choice of particular variables to those of the chosen robot, sensor or EDDI input topic.

The simplest option would be to consider the total fuzzing operation count in the test configuration as the value for a single dimension. But this considers totally distinct operations such as delay or value alteration to be equal, when they are not equal semantically. An alternative approach is that the fuzzing operation/variable combination can be treated as a bit set, in which the presence or absence of an operation consists of setting a particular bit. This would produce $2^{(O*M)}$ possible values - where O is the number of operations, M is that maximal number of instances. A complexity for this case is that the distance metric for this dimension would then be non-linear, with the distance being based on the “bit difference” between the bit sets for the two testing configurations.

Therefore, under SSPR a slightly different approach is used, in which one dimension for added per fuzzing operation, with the dimensional value being the normalised count for each fuzzing operation:

$O1 \leftarrow$ the number of RandomValueFromSet fuzzing operations in the test

$O2 \leftarrow$ the number of PacketLossNetworkOperation or BlackHoleNetworkOperation fuzzing operations in the test

$O3 \leftarrow$ the number of LatencyNetworkOperation fuzzing operations in the test

The purpose of this compromise approach is to consider each operation distinctly, since only fuzzing operations of equivalent type will be grouped and counted together. Although the operations chosen are not necessarily equivalent in terms of their impact on the simulation (due to their distinct parameters, and activation) their underlying semantics are much closer. It also ensures a linear distance metric for each dimension generated, avoiding the complexity of the proposed bit set alternative mentioned above.

4.2.2 Executing GA for Coverage Testing

The evolutionary algorithm uses the NSGA-II [22] algorithm with mutation and crossover to create new tests, discarding the worst performing campaigns from the population. The population is initialised with the randomised generation of fuzzing tests, with each test consisting of a series of operations. Following this, every generation is manipulated with mutation and crossover. Once evolution terminates, the testing platform produces an approximate Pareto optimal set of fuzzing campaigns, along with the associated mission requirement values. Intermediate results are also logged, so that users can examine the progression of the evolutionary algorithm, together with the time series of recorded metric values. Our approach also logs intermediate metric values (as a time series in files), depending on the configuration 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.

The version of the GA used in our preliminary version of the platform can be activated by selecting *NSGAEvolutionaryAlgorithm* from the DSL. This GA merely continues its evolutionary progress for a fixed number of generations. Our new coverage-aware GA (*NSGACoverageWithCells*) provides coverage checking, specifically, how well the parameter space of fuzzing configurations is explored, by partitioning into specific cells and checking for the execution tests covering a minimal proportion of cells.

The parameter space used is a user-chosen subset of the dimensionality described in SSPR in Section 4.2.1, with every dimension partitioned into a specified number of cells, spaced evenly between the minimum and maximum parameter number. The *dimensionRecords* attribute and the *coveragePerCell* of the *NSGACoverageWithCells* class can be used to define these parameters by the DSL, as shown in Figure 4.

Using SSPR, a particular fuzzing test configuration can be reduced to a single point in this space, and the corresponding cell identified. It is important to consider that we may only have low granularity for some of these dimensions, with some of them partitioned into a fixed number of cells, according to the importance of their values semantically. For example, dimension T3 of timing variance may be less important than T1 of midpoint mean, and therefore may be split into fewer, larger cells. If a value below or above the anticipated maximum or minimum cell boundary for a particular dimension is obtained, it is truncated to the extreme cell value.

The *NSGACoverageWithCells* class checks for coverage as part of its termination condition. After every generation is processed to produce offspring, dimensional reduction via SSPR is used to assign every one of them to a particular cell. Each individual configuration is then registered upon that cell. The proportion of cells that meet the minimal coverage criterion is estimated. If this proportion exceeds the minimal coverage level, then the algorithm terminates. There is also a failsafe value that is used as a termination condition if the algorithm does not achieve coverage.

Coverage Boosting With Genetic Algorithm Modifications. GAs are typically elitist and focused on the exploitation of the best results, e.g., with their selection operators choosing the best results from the Pareto front in order to produce new individuals. Mutation and crossover are typically performed randomly, which may have the drawback that common regions in the parameter space are explored repeatedly. A potential alteration is to use a *coverage-boosting* GA which seeks to take into account during evolution the coverage presently achieved, and how it can be increased.

This coverage-boosting GA can be implemented using the SSPR dimensionality reduction speculatively, with modified mutation operators. When performing mutation, every G generations, a proportion M_{CB} of the mutations are chosen for coverage-boosting mutation. This coverage-boosting mutation performs the standard mutation operator and, before evaluating it, checks whether it corresponds to an underserved cell not meeting the coverage criterion. If so, it is accepted and used as the output mutated individual. If not, then the process is retried up to T times, repeatedly altering the configuration with the mutation operator, until an underserved cell is found. This will necessarily reduce the elitism and likely therefore the performance in terms of metric optimisation, but it can improve the coverage that is achieved. It is also only available for the time-based fuzzing variant, as it requires the ability to assign a configuration to a cell before evaluating it, which is not possible for condition-based fuzzing.

4.3 Physical Testing Safety Considerations

Safety is a very important consideration during the physical testing of a scenario. Simulation allows fast execution of potentially destructive tests, e.g., robot intersection with physical objects/humans without any damage to either. However, some simulated tests may not be viable to execute directly. This may occur since either the modifications which they introduce to physical system values could cause system instability, or the operations themselves could disable sensors or override motion in ways that cause a collision or other risk. It is important to validate conformity between simulation with reality without risking damage to the scenario under physically or creating a hazard during the testing process. Several ways exist for doing this:

Safety Zone Expansion If inserting safety zones around a particular component is possible, the safety zones selected can be widened before simulation begins. This will ensure that when physical testing begins, human supervisors or automated interlocks can respond in the case of an imminent collision. The correct degree of expansion selected will depend on many factors, for example, the speed of motion of robots, and any worst-case delays in human or automated system response time. This will transform, for example, direct collision testing into close approach testing (in regards to the original safety zone).

Automated Safety Interlocks Safety interlocks (which for example control the maximum value of particular robot motion parameters such as joint values), or control and prevent entry to a specific region when a sensor is triggered are often employed to prevent damage to the MRS. When testing the implementation of control software in a situation in which physical safety prohibits entry to the active zone, triggering independent interlocks can be considered as an equivalent to physical violations. This strategy is adopted by our use case partner KUKA to perform part of their acceptance tests. It is important when relying on this, that the safety interlocks are completely independent of the fuzzing injected, and cannot be affected or disabled by the ongoing injected fuzzing.

One potential issue in this case is that simulation and the physical lab scenario may no longer be anticipated to produce values of equivalent magnitude. For example, consider a situation in which the simulation is performed to assess the number of time intervals in which a robot component intersects with a forbidden safety area. But in physical testing, an interlock is used to prevent entry to the selected area. When triggered, the interlock may automatically stop the ongoing scenario and return the robot to a known safe state, aborting further execution. Therefore, the violation metrics would not be expected to be equivalent between simulation and reality, but merely the presence or absence of a violation will become the criteria tested in checking conformity between simulation and reality.

Human Operator Intervention Similar to the case with the automated interlocks, it is possible to introduce human operators who can monitor the MRS. Together with information about the nature of the system and the presence of multiple interactions, they can take into account an impending collision and intervene to prevent it. For example, in the case of a UAV-focused scenario such as in the DKOX or KIOS use cases, it would be possible to introduce a safety pilot who can override the system (using a separate RC channel) and prevent a crash, close approach or other dangerous situation. An example of this was exploited in our earlier SAFEMUV project [74] in which a safety pilot was required during fuzz testing to ensure conformance to regulations to fly within an airport, and in order to operate fuzzing tests within a lab or final deployment scenario environment. The safety pilot, issued with a briefing about the hazards presented in the mission and the fuzzing operations and their anticipated effects, would be responsible for taking over control to prevent hazards when fuzzing causes an impending risk.

Virtual Objects In a test which involves a potential collision of a robot with an object, the physical test can be performed safely by removing the target object from the scenario during physical evaluation. By recording the position trace of the robot during physical execution, the geometry can later be analysed to determine whether an intersection with the zone of the virtual object occurred. This testing strategy was again employed in our earlier SAFEMUV project [73], in which the coordinates of the physical object (an aircraft wing) were stored before removing the physical wing from the lab scenario. Later examination of the trajectories traversed during the lab showed that the fuzzing applied would have caused a collision with the wing if it was not removed before testing.

Multi-robot Position Integration In a more complex variant of the virtual object scenario, the "virtual object" approach can be extended also to other robots. Consider a test involving the collision of multiple robots R_1 and R_2. The multi-robot position integration strategy would perform these tests twice with the same fuzzing configuration, firstly with R_2 artificially disabling itself and returning to a safe position for the relevant portion of the test (in advance of the collision). Secondly, an equivalent test will be performed with R_1 disabled. Replaying the position traces from both experiments, intersection testing can determine whether there would have been a collision or unacceptably close approach in a real system test.

4.4 Subset Selection For Physical Testing Selection

At this point in the process, simulation-based testing has been completed, potentially using the coverage-aware GA specified in Section 4.2.2 to enhance the coverage of the parameter space. This has considered S fuzzing scenarios during this process. Of these, S_{PARETO} constitute the final Pareto front for the selected metrics, given the maximal metrics for the selected experiments. A set of obtained simulation results providing observed timings (start and end of each fuzzing operation), and a set of result metrics for each of these configurations. At this point in the process, the goal is to select a subset, $P \subset S_{PARETO}$ of configurations for physical testing.

In this process, two conflicting objectives exist for the selection of configurations. The first potentially conflicting objective is *exploitation*: system testers wish to sample configurations with the highest metrics (e.g., most violations of safety parameters). This is useful in order to provide the set of physical testing results that would be most useful in terms of exposing the violations that can lead to design improvements to the MRS. A secondary purpose here is to ensure that these “most useful” fuzzing configurations can be reproduced under physical execution.

The second potentially conflicting objective is *exploration*; namely, providing wide coverage of the fuzzing parameter space in the physical results. Since the simulation is subject to reality gaps that lead to divergence between simulation and real performance, it is possible that more violations will be found during physical testing than expected from simulation (or vice versa), or that the distribution of these violations will be different in physical testing. It is also possible that these gaps may be non-uniform across the parameter space, so the parameter space of potential fuzzing operations needs to be widely sampled in physical testing to estimate the degree of conformance between simulation and reality. Considering both objectives, potential conflict between exploration and exploitation occurs since the results with the maximal metrics may be non-uniform across the fuzzing parameter space. For example, a reality gap may exist when a particular operation with a certain minimal parameter is used, although this operation is not used in the best configurations with the extreme values of any target metric. Therefore, sampling of the best results (pure exploitation) may not be prudent or wide enough to quantify the reality gap, and a balance between both is therefore required.

4.4.1 Exploration: Parameter Reduction And Distance Function

This uses the SESAME Standard Parameter Reduction (SSPR) strategy, as specified in Section 4.2.1, to reduce the complexity of the configuration to a set of values in the multidimensional parameter space. Following the completion of this, a distance metric can be defined to determine the separation of individual tests within the parameter space. The motivation for this distance metric is that determining the distances between points in the selected configurations can provide insight into how distinct the points are, and for a set of points, how effective the exploration provided by these configurations. Using the SSPR, the distance between two configurations u and v, in the SSPR can be computed by Equation 1. $Dims$ is the set of dimensions used (which may be a subset of all dimensions available in SSPR). The equation computes the Euclidean distance across all dimensions in $Dims$, since the space provided is continuous in all dimensions (adjacent points represent semantically similar configurations).

$$D_f(u,v) = \sqrt{\sum_{D \in Dims} (D(u) - D(v))^2} \tag{1}$$

4.4.2 Exploitation: Metric Quality Values and Metric Ranking

In order to exploit the results in the most efficient way, it is important in providing metric quality rankings, using an ordering function $MR(c)$; which given a fuzzing configuration $c \in S$ that has been evaluated in an experiment, emits a value in the range $[0, 1]$ to specify how interesting c is in respect to its output for physical evaluation. When there is only a single output metric, and the concern is for the maximisation of safety requirement violations, it is easy to achieve this by using this metric value directly to perform the sorting, and then normalising the results in the range. However, when multiple output metrics are used in ranking solution quality, system testers would have to determine which of the output metrics are most important for their purposes, providing a sorting that takes multiple objectives into account. The results would be sorted by the most important metric, with the second most important as a tiebreak when the most important is equal, in descending order of importance of the metrics for physical testing. Further, if users are interested in physically testing scenarios in which violations just above or below a certain range are obtained (such as in borderline scenarios where, for example, a near miss occurred), then more complex sorting functions can be used (for example, sorting a metric by its distance from a target violation level).

4.4.3 Costs of Performing an Experiment

A variety of costs could be incurred when performing an experiment. These costs are not necessarily financial, but may refer to the overheads (e.g., staff time) incurred to set up experiments, to execute them in a way meeting the safety constraints mentioned in Section 4.3, and to analyse or interpret the results obtained and assess the reality gap. Given the different nature of the experiments, these costs would likely be non-uniform, with some experiments requiring increased setup time, or extra time and effort to interpret the results. Therefore, every fuzzing configuration $c \in S$ can be assigned a $Cost(c)$ that determines the effective cost of performing that experiment. During the physical subset selection, this cost information will be used to determine how many fuzzing configurations can be tested given the available budget (e.g., effective testing time). If a particular configuration has properties that make it unsafe for physical testing and system testers would like to exclude it from consideration, then it can be set to a high cost exceeding the maximum cost budget.

4.4.4 Physical Subset Selection

This conflict between exploration and exploitation can be modelled as a max sum diversification problem [31], which permits system testers to control the tradeoff between these two parameters. This problem provides as an interface the configuration parameter λ, which manages the tradeoff between exploration and exploitation. At $\lambda = 0$, the selection algorithm only considers in physical testing the "best" simulation results, regardless of their diversity (spacing) in the parameter space. At $\lambda = 1$, only coverage (diversity) of the parameter space is considered, and the algorithm will select all scenarios equally regardless of the simulation result metrics obtained. Otherwise, a tradeoff between them will be obtained, considering the relative weight assigned to the hyperparameter λ.

The data inputs to the physical subset selection algorithm are presented below:

- Simulation results are provided in set S, containing $|S|$ configurations $c_0, c_1, c_2, ...$
- Maximum cost budget for testing resources: $MaxCost$
- λ value to define the tradeoff between coverage and exploitation for best results
- Metric ranking $MR(c)$ defined in Section 4.4.2
- Parameter distance function $D_f(u, v)$ - defined in Section 4.4.1
- Number of SSPR dimensions chosen for experiment - Dim_count
- Cost function $Cost(c)$ that determines the cost of performing an experiment - defined in Section 4.4.3

The algorithm outputs a subset, $BestP \subseteq S$, which contains the configurations selected for physical testing. The algorithm is presented in Figure 1. It considers potential subsets from the Pareto front, filters them for conformity to the maximum cost constraint, and then computes a value function using an equation that balances exploration and exploitation. This value function provides a linear combination of the metric rankings of included configurations, and the normalised distance sum between all included configurations, using λ as a scaling factor to control the relative preference of both of these terms (exploration and exploitation). The algorithm returns the subset found with the highest value.

Algorithm 1 The physical subset selection algorithm

```
$S_{Pareto} \leftarrow$ the set of points of $S$ on final Pareto front
$BestP \leftarrow null$
$BestValue \leftarrow 0$
$PS \leftarrow PowerSet(S_{Pareto})$ (giving all possible subsets)
for candidate set $PC \in PS$ do
    if $\sum_{c \in PC} Cost(c) <= MaxCost$ then
        Compute $DistSumNorm(PC) = (\sum_{u,v \in PC} D_f(u,v))/\sqrt{(Dim\_count)}$
        Compute $Value(PC) = (1-\lambda)(|PC|-1)\sum_{u \in PC} MR(u) + 2\lambda DistSumNorm(PC)$
        if $Value(PC) > BestValue$ then
            $BestP \leftarrow PC$
            $BestValue \leftarrow Value(PC)$
        end if
    end if
end for
Return $BestP$
```

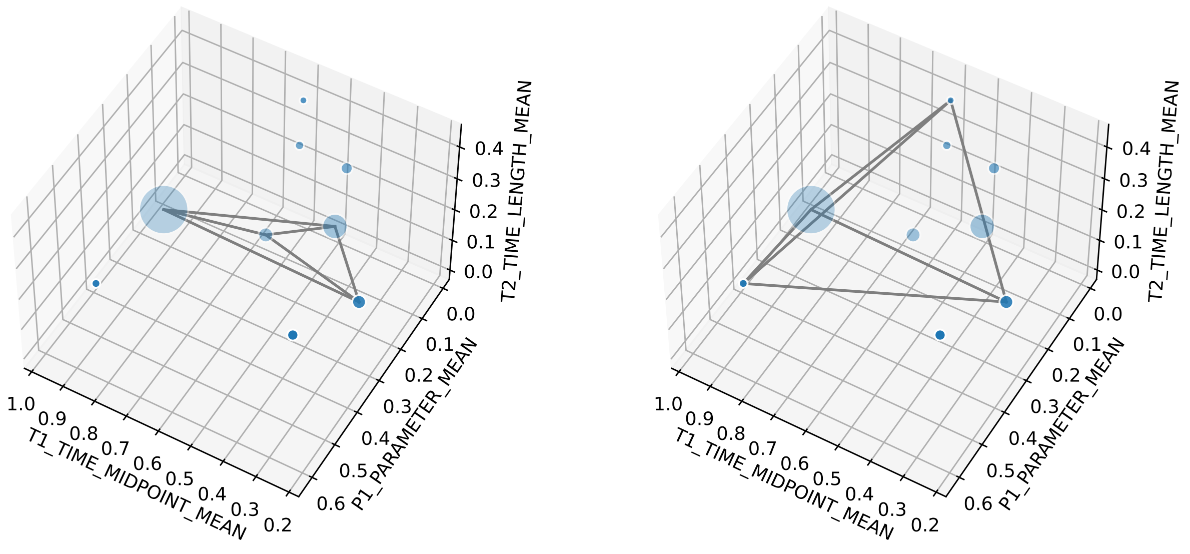
An example execution of the algorithm is presented in Figure 8. The points illustrated are from the Pareto front of an example GA execution, aimed at maximising coverage of the three dimensions labelled on the axes. The size of a point indicates the values of configurations under the metric ranking function MR, with the largest points having the highest metric values. $MaxCost$ is set to 4, and the four configurations selected are indicated with lines between them. Note that in practice full exploration of parameter spaces, e.g. considering every dimension of SSPR would involve eight dimensions, making its direct visualisation difficult.

It can be seen that the configuration parameter λ, manages the trade-off between exploration and exploitation. At $\lambda = 0$, the selection algorithm only considers in physical testing the "best" simulation results, regardless of their spacing in the parameter space (Figure 8a), and, consequently, all the selected configurations are relatively close and focus upon point with high metric values. At $\lambda = 1$, coverage of the parameter space becomes dominant in the selection, and the algorithm will select all scenarios equally regardless of the simulation results (Figure 8b), which leads to the selection of more widely distributed points.

4.5 EDDI Behaviour Testing

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 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.

The simulation-based testing platform can also be used to determine if the runtime EDDI is correctly acting as a reliable runtime monitor, identifying the violations of the safety or security conditions it is intended to detect. If a mitigation mechanism is available, our platform can be used to verify that the EDDI triggers the expected mitigation for one of these hazardous scenarios, potentially, when fuzzing occurs in addition to the intended attack. The EDDI runtime executor presents a generic interface which represents its communication in the form of simulator variables, integrated with the scenario-specific interface of the system. Thus, EDDI input or output messages can be modified via the standard MITM architecture to trigger these violations. Also, simulation-based testing can be used to fuzz the inputs to the EDDI, examining how its decision may be impacted or distorted due to the fuzzing information provided.



(a) $\lambda = 0.0$ (b) $\lambda = 1.0$

Figure 8: Examples illustrating subset selection and the impact of the hyperparameter λ

Furthermore, conditions of potential vulnerabilities identified from the fault trees in deliverable D4.3 [50] can be used in combination with our condition-based fuzzing approach, to inform the space of criteria for activating fuzzing. For example, in the power station inspection use case from KIOS/CCD, if fuzzing errors in human or robot localisation are applied at the time the EDDIs is supposed to trigger a secondary robot to assist another team member, a false negative can occur if the system does not perform this mitigation due to transient localisation errors. This integrated testing approach for the EDDIs together with the simulation can therefore concentrate on the conditions leading to the most severe hazards in order to use the simulation-based testing time most effectively.

4.6 Reality Gaps Detection and Exploration

A difficulty of simulation-based testing is that physical testing may produce different results from simulation. This may manifest as a test producing violation(s) in simulation which do not occur physically (false positive) or, more problematic for safety analysis, a test scenario may produce no violation(s) in simulation yet violations do occur physically (false negative). Further, the magnitude of violations produced may differ. For example, large violations may occur in a physical scenario which only produced small violations in the real experiment.

4.6.1 Local Exploration of Reality Gaps

It is possible that the first evaluations of the selected configurations may reveal reality gaps. The first approach in the presence of a reality gap would be the repeated execution both in simulation and physically the selected configuration (up to the limits of the execution budget in both configurations). This would allow users to quantify the determinism of both physical and simulated configurations, and determine if the gap still persists between the distributions under repeated execution.

If the gap identified appears to be persistent and stable, local exploration of regions of the fuzzing space can be used, to determine if the gaps are really persistent. The use of distance metrics in the subset selection allows other "close" configurations to be selected, by choosing to vary a parameter upon a particular dimension. This will help to isolate if the reality gap is entirely localised, or whether a critical parameter change can remove or reduce the gap. Note that the selected “close" configurations here for reality gap testing may not be from those identified from subset selection; they may merely be the closest in the simulated results to the configuration exhibiting the reality gap. It may also be possible to use statistical techniques to quantify the expected properties of any reality gaps (e.g., with the uncertainty about the expected gap reducing as more experiments are performed).

4.7 Reality Gap Compensation

Once a reality gap(s) has been identified, a useful approach for SESAME is to consider how they can be reduced. The first approach is by system/model identification, which attempts to solve the mismatch by improving the model of a particular component or components within the simulator. For example, if time series analysis of low-level traces shows that a controller is slower to respond in reality than under simulation, then a straightforward response is to add additional delays to the simulated model, until the reality gap at the low level is closed and the traces align properly. A localised search over the space of parameters can be performed to properly identify these values. Following this, the loop for GA execution can be repeatedly executed (closing the loop on the right-side of Figure 1) to determine if this leads to a better match between the evolved configurations and the observed physical system.

An alternative strategy that can be employed is randomisation. This may be especially useful if the system contains a learning algorithm which has over-optimised to specific inaccuracies of a simulated model. Domain randomisation could also prevent the SESAME evolutionary search process from seeking regions of the parameter space that result from reality gaps, due to simulated inaccuracies generating high metric values that necessarily disappear when tested in reality.

In order to perform domain randomisation, a potential approach is to sample a set number of simulation traces from the real system, and use their sampled variability in adding randomisation into the real system. This is a form of *noise-based randomisation* similar to within [78]: adding a certain level of noise to the simulated input values, in order to prevent the evolved fuzzing process from becoming over-optimised to particular features of the simulation. Following this, it is possible to go back to the GA Evolution stage of the methodology in Figure 1, re-run the simulated experiment with this modified simulation, and see if it exhibits the same reality gap. The changes to the simulation (with the injection of the simulated noise) may have re-adjusted the evolutionary process and changed the final Pareto front.

5 Implementation

5.1 MRS Testing Platform Implementation

This section presents an overview of the implementation of the simulation-based testing and new components relevant for physical testing components. Further details about the implementation of simulation-based testing were presented in [52], and interfacing and similar are presented in [57].

The code developed by the project WP6 will be released as open source upon project completion, and is currently available at `https://github.com/sesame-project/simulationBasedTesting`.

5.1.1 Simulation-Based Testing Implementation

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]. Apache Kafka[4] and Flink[5] are used to interconnect the MRS simulator, the individual test runners, and the experiment runner 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 to implement fuzzing operations.

The simulation-based testing implementation currently supports Linux and Windows operating systems. On Linux, shell scripts incorporting the Maven build tool [6] are used to recompile code components dynamically generated during the execution of the experiments. Kafka runs natively to provide the message bus support. On Windows, Cygwin is used to provide a virtual Linux environment for the execution of the associated shell scripts, and the Kafka environment is provided using a Docker container, as recommended for its execution on Windows. The simulation-based testing implementation carried out so far is structured as several inter-dependent Eclipse projects. All projects (except for the generator project) are Maven based, allowing their dependencies to be automatically downloaded:

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`. In addition it includes several key data types for system generic events (EventMessage, MetricMessage and ControlMessage) together with associated serialisation and deserialisation code.

uk.ac.york.sesame.testing.dsl: In this project, we store the models of the simulation-based testing DSL involved in the project. The current version of the Testing DSL described in Section 3.7 is stored as Emfatic (under TestingMM.emf). From this Java code representations of the associated classes can be generated by transforming this to an Ecore file and using Testing.genModel. This allows Java code to manipulate the models dynamically for new test generation and results processing

uk.ac.york.sesame.testing.generator This is an Eclipse plugin project which supports the generation of the necessary code for SESAME experiments. Several code generators (implemented in Epsilon Generation Language, EGL) are contained which produce Java code for launching test campaigns, together with TTS and ROS middlewares for test execution, and fuzzing operations with the intended parameters

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

uk.york.sesame.testing.evolutionary This package provides support for evolutionary experiments using the JMetal framework [42], supporting population generation, mutation and crossover operators and interfacing with the testing model, adding the newly generated Tests to the model. When condition based fuzzing is used, it also depends on support for JGEA [7]. This package stores the metrics communicated back from the model generated test runners back into the model under the relevant Tests. It also stores code for exporting the model information for the dimensionality reduction described in Section 4.2.1 (under the `phytesting` package.)

5.2 Simulator-Specific Interfaces

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. The architecture presents an interface, ISimulator, which users must implement to connect their simulator with the provided infrastructure. The package `uk.ac.york.sesame.testing.architecture.ros` contains an interface to ROS/Gazebo simulations, which implements ISimulator. Its simulation interface uses *jrosbridge*, which provides an interface using the standard package ROSbridge upon the MRS side.

The implementation for the TTS simulator used in the KUKA use case is presented in package `uk.ac.york.sesame.testing.architecture.tts`. The simulation interface in the class TTSSimulator, which also implements the generic interface ISimulator, is implemented using the gRPC protocol [32], and a protobuf protocol definition contained in *SimLogAPI.proto*. This protocol definition can be automatically converted into Java code and used by the TTSSimulator class to interface with the messages generated during the simulation.

5.3 Code Generation and Performance Metrics

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 instance of the SESAME Testing DSL, specifying the structure of the testing space, as defined in Section 3.7.

During simulation-based testing, metric templates and experiment runners 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 instance of the testing DSL. The metric template consists of numerous method hooks the user can implement to define the metric initialisation and processing in response to events. 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. System testers will implement the *processElement1* method to define the response of the metric to incoming events. An example of a completed metric for a case study for the KUKA use case is presented later in Section 6.1.5.

5.4 Test Generation

Both genetic algorithm implementations, the conventional and the new coverage tracking GA, are implemented using JMetal, with the standard class available in **NSGAII_ResultLogging** and the coverage tracking/boosting in **NSGAII_ResultLogging_Coverage**. These classes provide a slightly modified version of the conventional NSGA-II [22, 42] algorithm modified for logging of intermediate state and debugging. When the new coverage-tracking GA is specified, the nd4j library is used to track the percentage of implemented cells.

5.5 Subset Selection

The subset selection implementation is provided in Python, using among others the libraries *networkx* (for storing the relationships between configurations as a graph) and *matplotlib* (for rendering). After selecting the model file and an executed test campaign, the model is processed to perform the dimensionality reduction and write out raw dimensional values to a CSV file. From this, a Jupyter notebook can be loaded which reads this information and implements the subset selection algorithm described in Section 4.4. This allows interactively setting parameters such as λ and examining/visualising the selected configurations for physical selection.

6 KUKA Case Study

6.1 KUKA Test Cell Scenario

6.1.1 Introduction

In this section, we consider the application of our SESAME testing techniques to an example case study from SESAME use case partners. Future work during the project evaluation phase will consider its application to other use cases, testing the ROS-based interfaces. The KUKA use case is an industrial assembly case used for the manufacture of gearboxes (see deliverables D6.5 [56] and D8.8 [55]). This use case incorporates a combination of three robots with sliding conveyors that interact to transport the gearboxes between assembly robots, together with human workers surrounding the assembly cell who assist with specific tasks requiring manual intervention. The arrangement of the cell is depicted in Figure 9.

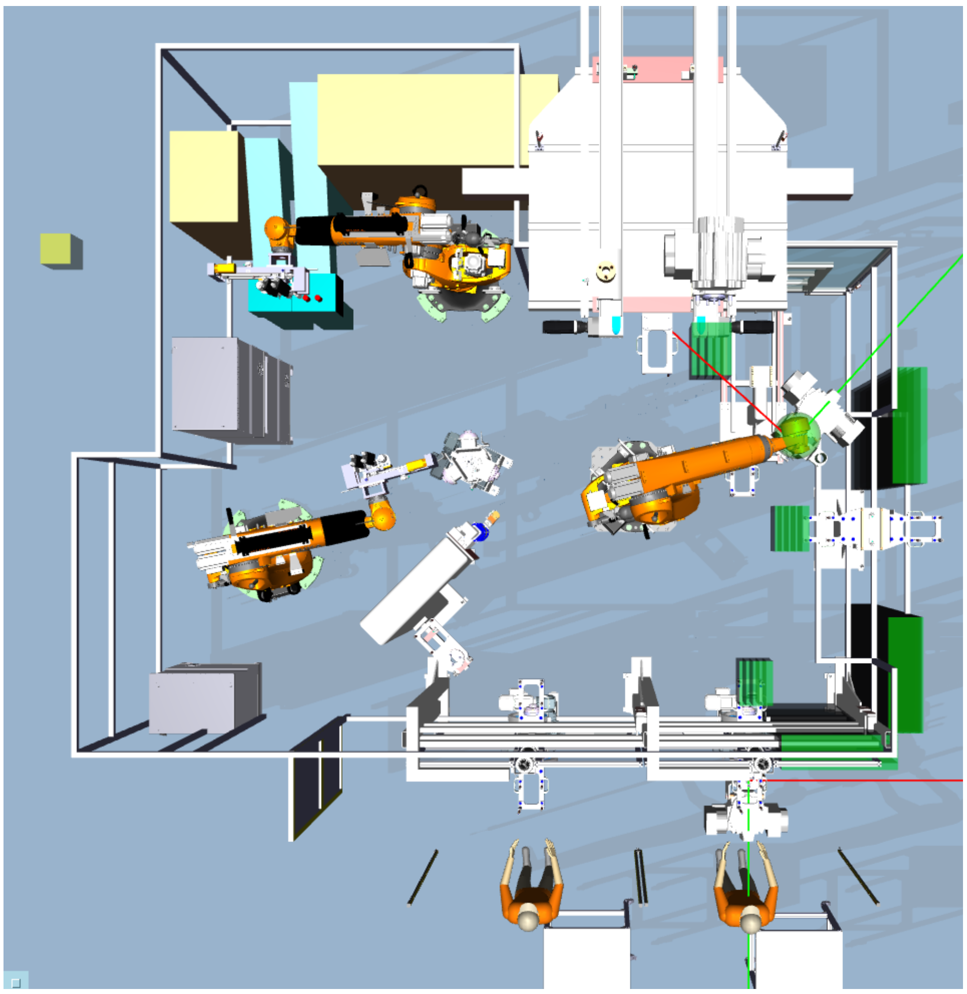


Figure 9: Screnshot of KUKA cell under simulation using the DDD simulator from TTS

The intent of this case study for the industrial partner (KUKA) is to achieve its functional requirements (processing the gearboxes at a consistent rate), while also ensuring non-functional requirements related to safety and turnaround time. In particular, it is important to guarantee the safety requirement that robots must not generate hazards to human operators during operation. This motivates a testing approach to validate candidate scenarios and discover situations in which fuzzing operations could cause hazards to the human operators or other staff working in the area. The primary safety requirement is to ensure that robots do not enter regions

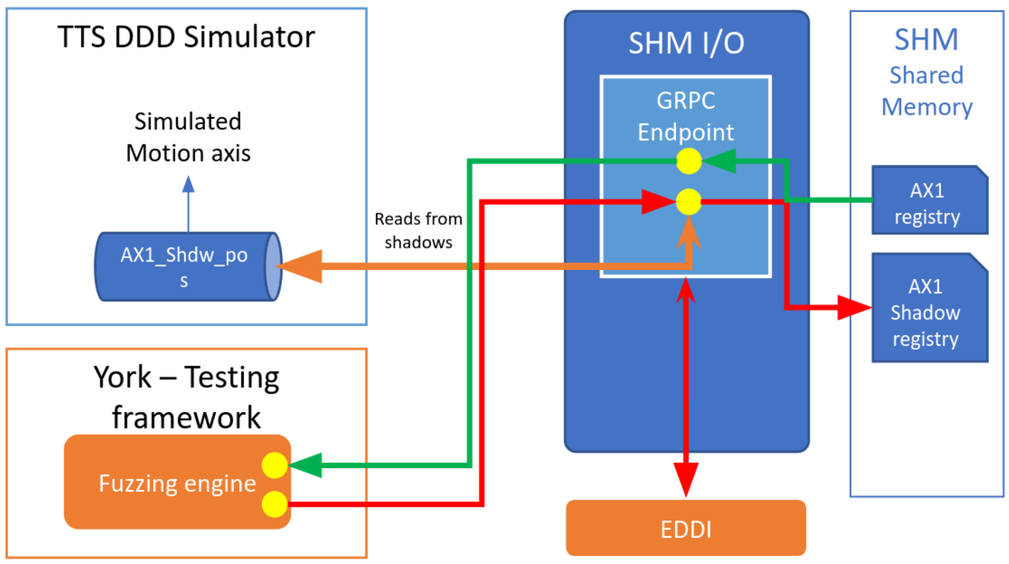


Figure 10: KUKA use case in the simulation-based testing (with DDDSimulator) scenario

that could cause a collision hazard to the human operator or other workers. Under normal circumstances, motion trajectories are pre-planned to ensure conformity to this safety requirement. However, runtime faults in communication, sensing or motor/interlock actuation may lead to robots malfunctioning and rotating in order to enter these forbidden areas. It is also important to expose situations in which the EDDI fails to operate in order to preserve correct operation.

6.1.2 Simulation Scenario

The simulation scenario was supplied for testing and was implemented using the TTS simulation platform [71], which includes an integrated DDD model editor allowing the cell to be modified and customised, e.g., to add safety zones. The structure of the simulation scenario is shown in Figure 10. Interconnection to the testing platform was accomplished using the gRPC protocol [32], passing messages to and from a shared-memory component which acts as a central point for state storage, and an intermediary to the SESAME simulation-based testing platform. Further information on our simulator-specific interface for TTS can be found in Section 5.2. Safety violations are tracked using a collision detection module in the underlying TTS simulator, which is capable of registering the intersection of the robot arm with various statically defined collision zones (cuboids). The simulator sends SafetyZone messages using the gRPC API when these zones are entered by a tracked object. It also supports dynamic inter-robot collisions. In some of the scenarios presented by our KUKA industrial partners, Simit is used to simulate the PLC code that provides the logical implementation. Further information on the structure of the scenario is available in the related project deliverable [56].

6.1.3 Physical Implementation

In the physical implementation, the robots are controlled by a PLC controller which sends motion commands to actuate their joints, and receives sensor data for updates on the action of the various robots and to synchronize their operations. When testing the system physically, the shared-memory component also communicates with the PLC, receiving their data and providing a common interface to the testing platform. In addition, the TTS DDDSimulator is still used, since it must subscribe to data received from the PLC and register collisions computed from the received joint values and simulation 3D model. The structure of the scenario in the physical implementation is presented in Figure 11. Particular differences that may exist between the simulated scenario and reality include increased latencies for communications between the PLC and the shared-memory component (to the order to 50ms) which may generate a reality gap between simulation and physical implementation during fuzzing. Also, according to consultation with simulation developers, the physics of the system is not entirely determinsitic - for example, variations in robot torque during motion may lead to discrepancies between the timing to reach a given waypoint in simulation versus reality.

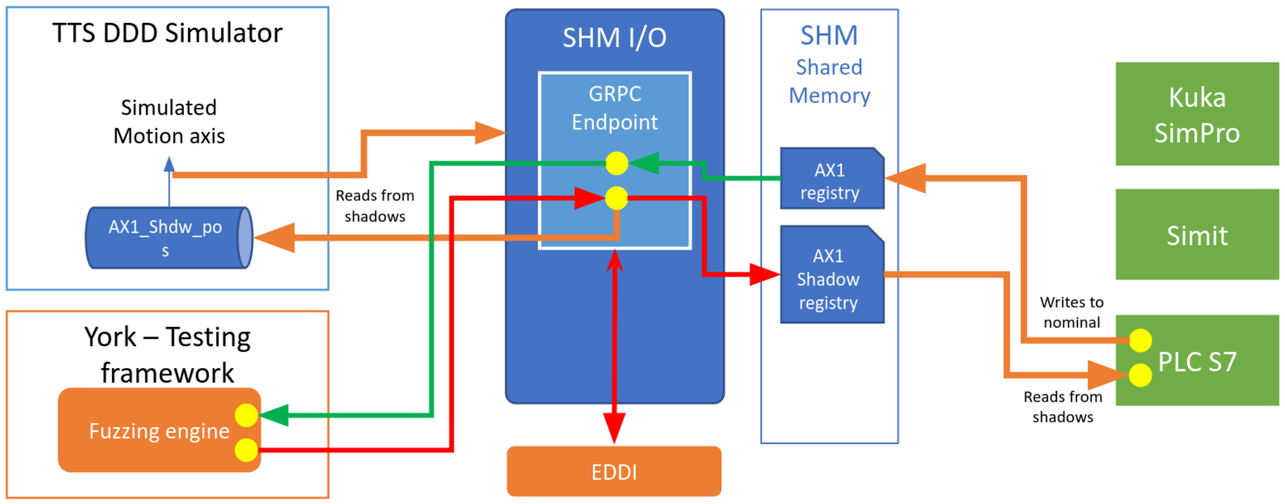


Figure 11: Architecture description of components interfacing with the physical system

6.1.4 Scenario Specification

The requirements for the safe operation of this scenario are presented in Table 2:

Table 2: Requirements of KUKA case study

ID	Description
R1	Robots must avoid hazards to human operators as a result of their gripper's collisions with the safety zones (green areas)
R2	Robots must move within their allowed speed limit
R3	Robots must avoid collisions or close approaches with each other
R4	The MRS must complete assembly of engines by the defined cycle time (performance requirement)

Following Step 1 from the methodology in Figure 1, after a safety analysis, we have identified the communication from the controller to the robot arm joints as vulnerable to message disruptions, and requirement R1 (collision with the safety zones) as the requirement to assess in regards to robots causing potential hazards to human bystanders. Therefore, requirement R1 will be tested in an initial evaluation of this case study, which requires specific support from the simulator with collision detection logic, to detect collisions between the sphere zone surrounding the gripper and the defined safety zone.

Fuzzing is applied by altering the target motion value for robot arm joints, with a random offset applied to the instantaneous joint values for $Link0$ and $Link1$ of R3200 to produce motion jitter around the intended joint value. Also, fuzzing operations of message deletion and delay will be applied to $Link0$ and $Link1$.

There are several possibilities that occur in considering methods for the safe execution of this cell in the physical environment. It can be ensured that the safety zones are virtual objects, so the robot can enter them

```java
public void processElement1(EventMessage msg, Context ctx, Collector<Double> out) {
    String completionTopicName = "safetyzone";
    String topic = msg.getTopic();
    if (topic.contains(completionTopicName) && topicMatches(topic)) {
        if (msg.getValue() instanceof String) {
            String s = (String) msg.getValue();
            Optional<ROSMessage> rosmsg_o = ROSMessageConversion.fromJsonString(s);
            if (rosmsg_o.isPresent()) {
                ROSMessage rosmsg = rosmsg_o.get();
                SafetyZone sv = rosmsg.getSafetyZone();
                float level = sv.getLevel();

                if (violationCount.value() == null) {
                    violationCount.update(0L);
                }

                if (level < getLevelThreshold() && isReadyToLogNow()) {
                    violationCount.update(violationCount.value() + 1);
                    out.collect(Double.valueOf(violationCount.value()));
                }
            }
        }
    }
}
```

Figure 12: The metric implementation for collision occurrence

without damage, and obviously during the execution of the tests, humans would be restricted from entering the area surrounding the cell. If extra assurance is required, safety interlocks can be used as an additional guarantee to ensure that the joint range of travel is limited. Any joint excess that would lead to penetration of the safety zone would be considered as a violation.

6.1.5 Metric Definition

Figure 12 presents a fragment of the implementation of the $collisionOccurrence$ metric used to quantify violations of R1 by tracking the number of intervals in which a collisions of the gripper safety zone with any safety zone. The metric operates as follows:

If the region surrounding the robot gripper (green sphere in Figure 9) collides with these regions, the collision detection logic will trigger a safety zone message which will be sent to the testing platform via the DDDSimulator custom API over gRPC. As an inbound simulator event, these will then trigger the processElement1 method. In this method, the metric inbound topic is verified to ensure they refer to safety zone messages. If the event is sufficiently significant, both in terms of the timing rate and penetration depth of the safety zone, the violation is recorded by incrementing violationCount. This value is produced as the output metric value, to be monitored by the test runner and its final value will be logged to the model as the output value of $collisionOccurrence$ for this simulation. The method *isReadyToLogNow* provides rate limiting; every 0.2 seconds it will trigger a violation which will be tracked and counted towards the safety violation metric. This rate limiting is used to provide independence from the event rate of the simulation.

6.1.6 Simulation Model and Fuzzing Conditions

An excerpt of the fuzzing model instantiated for the KUKA industrial use case is presented in Figure 13. Since all the safety zones are adjacent to robot $R3200$ (middle right) we concentrate our fuzzing upon this robot. The

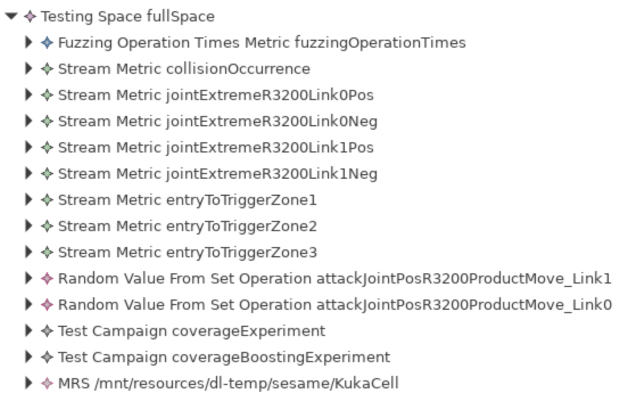

- Testing Space fullSpace
  - Fuzzing Operation Times Metric fuzzingOperationTimes
  - Stream Metric collisionOccurrence
  - Stream Metric jointExtremeR3200Link0Pos
  - Stream Metric jointExtremeR3200Link0Neg
  - Stream Metric jointExtremeR3200Link1Pos
  - Stream Metric jointExtremeR3200Link1Neg
  - Stream Metric entryToTriggerZone1
  - Stream Metric entryToTriggerZone2
  - Stream Metric entryToTriggerZone3
  - Random Value From Set Operation attackJointPosR3200ProductMove_Link1
  - Random Value From Set Operation attackJointPosR3200ProductMove_Link0
  - Test Campaign coverageExperiment
  - Test Campaign coverageBoostingExperiment
  - MRS /mnt/resources/dl-temp/sesame/KukaCell

Figure 13: A fragment of the DSL example for the KUKA use case showing selected fuzzing operations

operations $attackJointR3200ProductMove\_Link1$ and $attackJointR3200ProductMove\_Link0$ of the model illustrate this fuzzing offset value change, using the *RandomValueFromSet* fuzzing operation to apply relative changes to the joint value time series.

For the condition-based fuzzing approach, one approach considers the activation of fuzzing when extreme joint values are reached, which represent extremes of travel. For example, fuzzing may be activated when a robot joint reaches the upper or lower limits of its range of extension. The intent behind this is that these extreme positions may lead to instability or safety violation, such as the entry into the safety zone mentioned in requirement R1. Therefore, condition variables are included based on metrics that return a positive integer value when extreme values (positive and negative) sampled from a particular joint's time series are reached.

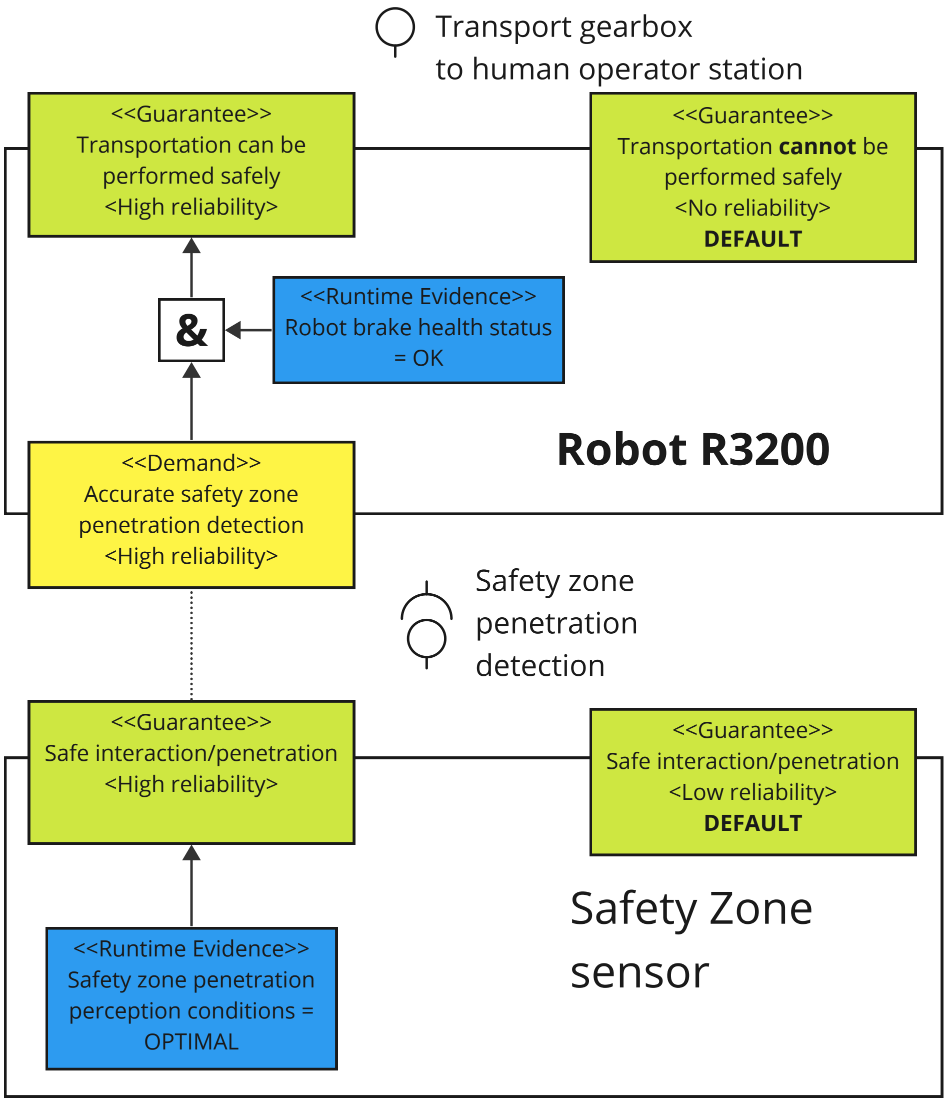
These condition variables obtain their values from the $jointExtreme$ metrics specified in Figure 13. For negative extremes of $Link0$ of robot $R3200$, the relevant metric produces values upon reaching a range of $[-\pi \rightarrow -2.5]$. For positive extremes of $Link0$ of robot $R3200$, the relevant produced values upon reaching a range of $[2.5 \rightarrow \pi]$. For negative extremes of $Link1$ of robot $R3200$, the metric produces values upon reaching a range of $[-0.4 \rightarrow -0.5]$. For positive extremes of $Link1$ of robot $R3200$, the metric produces values upon reaching a range of $[0.8 \rightarrow 0.9]$.

In addition to the extreme joint extensions used as triggers for condition-based fuzzing, there are three spatial trigger zones that can serve as triggers for activating this type of fuzzing. These will be accessed at different points of the trajectory of $R3200$: Trigger zone 1 is accessed when passing close to the right-hand surface carrying the gearbox; trigger zone 2 is accessed when approaching the conveyor to pick up the gearbox; and trigger zone 3 is accessed when placing it close to the workers for manual operations. The relevant condition variable produces positive variable values when the robot's gripper enters these trigger regions, and can serve to activate fuzzing at specific points in the operational sequence corresponding to reaching specific geometric zones. These trigger zones are the small cubic zones shown in Figure 9.

6.1.7 EDDI Specification

An EDDI - in this case, a Conditional Safety Certificate (ConSert) - is created to examine the motion of the robot from picking up the gearbox and transporting it to the human operator station at the bottom of the cell. The EDDI presumes that a safety concept, involving detecting proximity to the safety zone via external sensor, is in-place. When the R3200 penetrates the safety zone beyond some threshold, the sensor should notify the robot (directly or otherwise), and the R3200 should then immediately brake to avoid collision with the human operator. The EDDI acts as a runtime monitor above the safety concept, considering whether the sensor and robot can still guarantee the safety concept assumptions hold. This depends on the sensor being able to guarantee it can accurately detect safety zone penetration and on the robot being able to guarantee that it can apply sufficient brake force in case a non-trivial safety zone penetration happens. The EDDI runs as a separate process and is interconnected directly over gRPC to the shared-memory interfacing component, subscribing to the selected input variables and publishing back its decision of safe motion as a Boolean $EDDI\_SAFE$ variable.

The EDDI input data includes status notifications regarding the health of the R3200 braking actuator (e.g. low brake pressure notification) and of the safety zone sensor (e.g. low-luminance conditions detected). This information is relevant to the ’Runtime Evidence’ nodes indicating "Robot brake health status = OK" and "Safety zone penetration perception conditions = OPTIMAL" respectively. When the necessary conditions are invalid, the Runtime Evidence nodes are also updated to ’false’ in the ConSert, which propagates upwards to switch the top Guarantee to ‘Transportation cannot be performed safely’. A specification of the EDDI as a ConSert diagram (explained in full detail in deliverable D7.1 [53]) is presented in Figure 14. The ConSert operates in a similar manner to a Boolean logic formula, in which preconditions are evaluated hierarchically by logical operations to determine its instantaneous output. Considering this information, if the EDDI assesses that the motion cannot be completed safely, the $EDDI\_SAFE$ variable will be set to false, otherwise, if this guarantee can be provided given the evidence, it will be set to true.



Transport gearbox to human operator station
<<Guarantee>> Transportation can be performed safely <High reliability>
<<Guarantee>> Transportation **cannot** be performed safely <No reliability> **DEFAULT**
&
<<Runtime Evidence>> Robot brake health status = OK
Robot R3200
<<Demand>> Accurate safety zone penetration detection <High reliability>
Safety zone penetration detection
<<Guarantee>> Safe interaction/penetration <High reliability>
<<Guarantee>> Safe interaction/penetration <Low reliability> **DEFAULT**
Safety Zone sensor
<<Runtime Evidence>> Safety zone penetration perception conditions = OPTIMAL

Figure 14: EDDI definition for KUKA case study presented as ConSert sketch

6.2 Experimental Demonstration on KUKA Case Study

6.2.1 GA Performance Test

This stage corresponds to Step 3 of the integrated testing methodology of Figure 1, and Steps 3.3 and 3.4 of Figure 2. An evaluation is performed on the GA to test its performance in terms of coverage, on the example KUKA case study presented in this section. Both forms of the GA are evaluated, the coverage-tracking GA in which SSPR is used to reduce points to dimensions and monitoring the cells which are accessed (Section 4.2.2), and the coverage-boosting GA in which a custom mutation operator is used to increase the GA performance (Section 4.2.2). This is used for time-based fuzzing test (since this is currently the variant supported by the coverage-boosting GA). The RandomValueFromSet fuzzing operation is tested in this evaluation, applied to Link0 and Link1 of $R3200$. A subset of the dimensions from SSPR (defined in Section 4.2.1) are used in this test:

- T1 - the mean of fuzzing interval midpoint
- T2 - the fuzzing interval mean length
- P1 - the mean of all normalised fuzzing parameters

The parameters used are given in Table 3:

Table 3: Parameter values

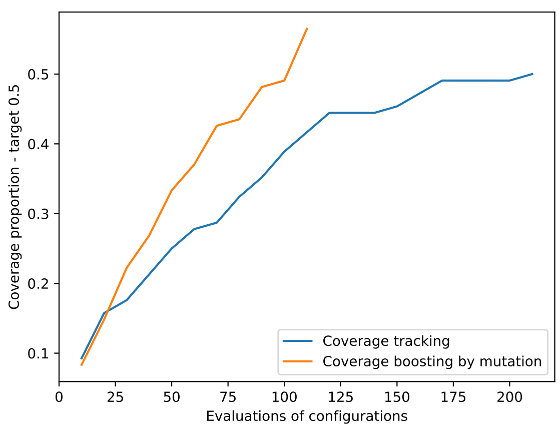
Population size	10
T1 dimension range; number of cells	$[0, 1]$; 6
T2 dimension range; number of cells	$[0, 0.5]$; 6
P1 dimension range; number of cells	$[0, 1]$; 3
Target coverage per cell	1
Target cell coverage proportion	0.5
Time length for simulation	70s

The comparison of the coverage achieved by both GAs over time (described in Sections 4.2.2 and 4.2.2) is presented in Figure 15a. The results show that the coverage-boosting GA is significantly faster at reaching the target cell coverage proportion, reaching at at 110 evaluations (or 11 generations), as opposed to 210 evaluations for the standard coverage-tracking GA. This consists of approximately three hours for execution of the coverage-boosting GA as opposed to nearly five hours for the coverage-tracking GA. This is due to the increased mutations which favour landing upon a more diverse set of cells, therefore increasing the coverage more quickly.

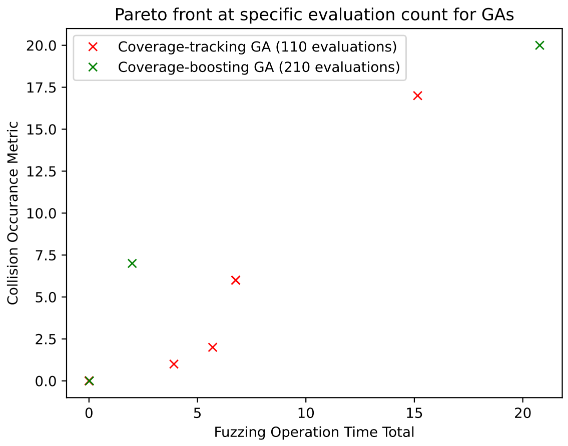
In order to examine the impact on the output Pareto fronts from both GAs (coverage tracking vs coverage boosting), Figure 15b shows a comparison of the Pareto front of output metrics for both the coverage-tracking and coverage-boosting GAs. The horizontal axis shows the fuzzing time total for a configuration, and the vertical axis the output value for the collision occurrence metric. When this is done, the results show that the coverage-boosting GA also found some improved results with improved metrics (a short fuzzing configuration with time total approximately 2, but with a collision occurrence metric of 7.5). There is also an extreme value with fuzzing operation time around 20 seconds and a collision occurrence at 20. Although there are a lot of configurations on the front with different parameters and repeated time values, since the incentive of the GA to explore parameters sometimes reduces the number of distinct points on the front.

6.2.2 Physical Subset Selection Example

This stage corresponds to Step 4 of the integrated testing methodology of Figure 1; the selection of a subset of results from the simulation-based testing output for physical testing. When output Pareto fronts from



(a) Coverage of different GAs



(b) Comparison of Pareto front for different GAs

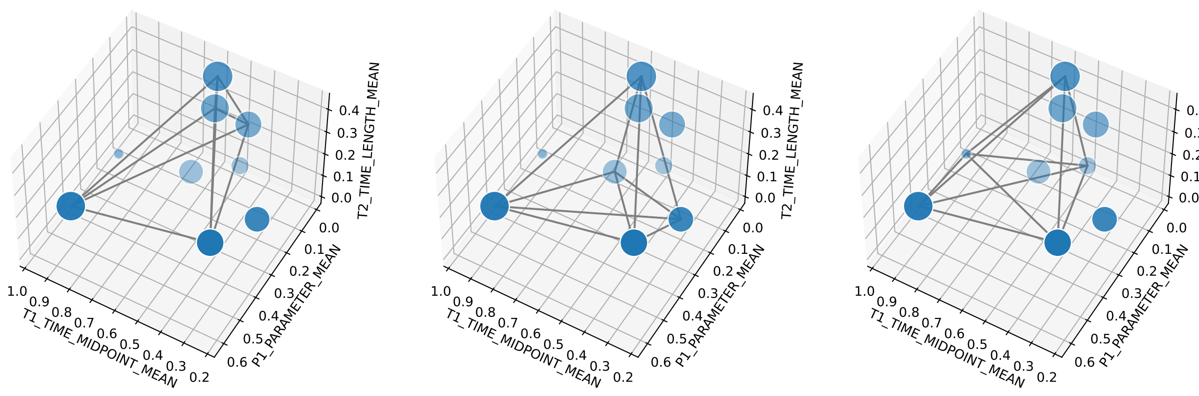
Figure 15: Comparison of GAs

simulation-based testing only contain a small number of configurations, the physical testing cost budget (Max_Cost) may be sufficient in order to test all of them. However, when the results sets contain increased diversity and a larger number of configurations, it is possible to use the subset selection algorithm in Section 4.4. Here we use an alternative example with more diversity and additional points in the Pareto front, obtained from a seperate execution of the time-based fuzzing GA in Section 6.2.1.

We assume that the maximum cost budget permits the evaluation of 5 configurations, and that all configurations are assumed of equal unit cost. The input set of configurations from the Pareto front considered are presented in Table 4, and the subset selection examples are presented in Figure 16. The configurations show the selected points in the 3D subset of SSPR space. The metric quality is assigned from the value for collision metric, with the largest values (most violations) illustrated with the largest points. When $\lambda = 0$ the subset selection finds the largest metric values (with every selected configuration having a collision occurance metric at least 9), which is useful to exploit the highest metric values. With $\lambda = 1$, as expected, there is a much wider range of output metrics in the physical testing scenarios, including two with collision occurrence metrics of 0 and 2 (a no and low violation case). Testing the zero value configuration serves as a validation of whether there is a reality gap in the no violation case, that is, whether this is still safe under no scenario. We can also see that the subset selection picks two tests in common over the range of λ values (Test_158 and Test_161) which have some of the highest lambda values. Overall, the tests in the input Pareto front that have high metric quality are relatively well distributed throughout this parameter space considered, which accounts for the commonality.

Table 4: Structure of the Pareto front and the test configurations for time-based fuzzing

Configuration	FuzzOpTimes	collisionOccurrence	Operations and Parameters
Test_103	0.003	0.000	1:Link1([0.14-0.22])
Test_158	49.238	18.000	2:Link0([0.27-0.29]),Link0([0.27-0.30])
Test_027	32.993	14.000	2:Link0([0.27-0.29]),Link0([0.27-0.30])
Test_174	15.089	10.000	1:Link1([-0.10-0.17])
Test_161	34.720	15.000	2:Link1([-0.17-0.12]),Link0([-0.30-0.07])
Test_209	2.351	6.000	1:Link1([0.11-0.17])
Test_201	13.203	7.000	1:Link1([0.09-0.27])
Test_187	1.981	2.000	1:Link0([0.29-0.30])
Test_137	13.480	9.000	1:Link0([0.29-0.30])



Test_027	Test_158	Test_103
Test_137	Test_161	Test_158
Test_158	Test_174	Test_161
Test_161	Test_201	Test_174
Test_174	Test_209	Test_187
(a) $\lambda = 0.0$	(b) $\lambda = 0.5$	(c) $\lambda = 1.0$

Figure 16: Subset selection for GA example with different λ values - including the selected tests in each subset

If all the configurations with high metric output were closely clustered in one region, the trade-off between exploration and exploitation would be less pronounced. But in this case, even when changing the trade-off between exploration and exploitation, there is still some commonality to the choices made.

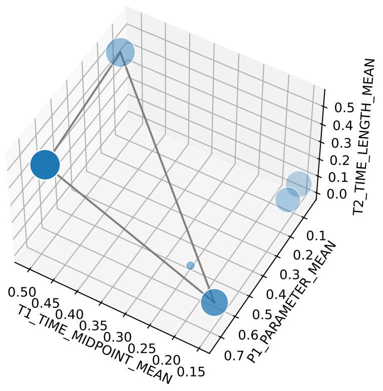
6.2.3 Condition-Based Fuzzing

Condition-based fuzzing was also tested with the coverage-tracking GA. The coverage of the fuzzing experiment was significantly lower than in the time-based experiment. At an equivalent point in the evolution to the previous time-based fuzzing test (210 evaluations), the coverage achieved is 0.102. However, this lower coverage may be due to the condition-based fuzzing only allowing particular timing points for activation and deactivation, as specified by the semantics of the conditions, so in condition-based fuzzing, the conditions have to trigger activation evenly distributed to achieve full coverage. Table 5 shows the Pareto front configurations for condition-based fuzzing at an equivalent point in the evolution as time-based (210 evaluations), and shows that the condition-based fuzzing did find some interesting configurations. It shows that the condition-based fuzzing found a large collision metric value (19), but with a significantly longer fuzzing time. Some of the shorter fuzzing configurations (around 5 seconds), which produced short violations, may be interesting to investigate physically. The conditions used that produce the highest metrics trigger fuzzing generally upon the joint extremes, particularly upon the negative extreme of Link1, or upon the first entry to trigger zone 2.

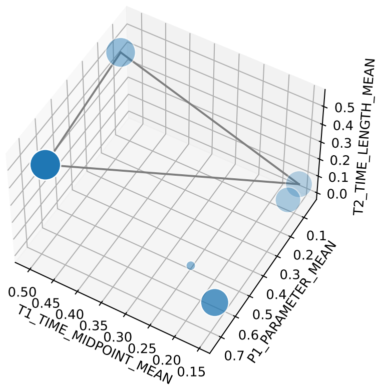
Since the front only contains six points, the subset selection was this time demonstrated with Max_Cost permitting the selection of three configurations, again assuming all configurations are of equal unit cost. The set of configurations from the Pareto front obtained from simulation-based testing are shown in Table 5, and the subset selections at various λ values are presented in Figure 17. With $\lambda = 0$, three configurations that produce the highest metrics are selected, two long timings and some very short. As λ is increased to 0.5, Test_146 is substituted for Test_045. With $\lambda = 1.0$, two of the lower collision metric configurations (Test_146 and Test_045) are included in the selected set.

Table 5: Structure of the Pareto front and the test configurations for condition-based fuzzing

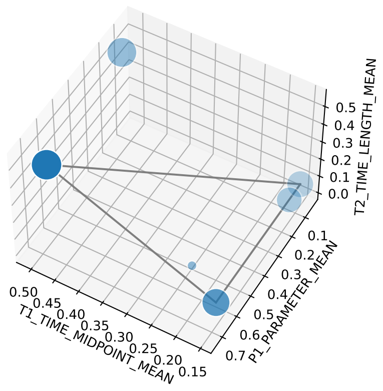
Configuration	FuzzOpTimes	collisionOccurrence	Operations and Parameters
Test_006	35.950	19.000	1:Link1([-0.19-0.25])
Test_104	32.360	7.000	1:Link1([0.17-0.25])
Test_045	5.050	3.000	1:Link0([-0.10-0.29])
Test_146	4.900	2.000	1:Link0([0.01-0.04])
Test_093	4.890	1.000	1:Link0([0.07-0.15])
Test_190	0.000	0.000	1:Link1([0.08-0.24])



Test_006
Test_045
Test_104
(a) $\lambda = 0.0$



Test_006
Test_104
Test_146
(b) $\lambda = 0.5$



Test_006
Test_045
Test_146
(c) $\lambda = 1.0$

Figure 17: Subset selection for GA example for condition-based fuzzing with different λ values - including the selected tests in each subset

6.2.4 Physical Testing Safety Issues

At this point, subset selection has been completed from the simulation-based testing results, to find configurations for physical testing, and physical testing of these selected scenarios can be performed as described in Step 5 of the methodology. In order to test these scenarios upon the case study, it is necessary to consider physical testing safety, using mitigations such as those discussed in Section 4.3. One possibility is the use of automated safety interlocks or human operator intervention to prevent any potentially hazardous impact during collision testing, although the thresholds for action would need to be calibrated to ensure that any collision with the safety zone would be equivalent to triggering the interlock. In this case, reality gap testing would consist of merely checking the presence or absence of a violation under physical testing, not its magnitude. A further alternative could be similar to the “virtual objects” technique of Section 4.3; to modify the physical structure of the scenario (by removing any physical obstructions surrounding the safety zones) so that entering the safety zone would be possible without risk during physical testing.

7 Conclusion

This report presented the various strategies that are used to meet the challenges presented for transitioning from simulation-based testing towards lab and physical testing in Section 1.1.

We propose the use of simulation-based testing integrated into a structured methodology allowing the transition towards physical testing. Following the specification of the scenario, its requirements, simulator support components and appropriate fuzzing configurations, a simulated testing campaign can be launched involving evolutionary algorithms which support either time-based or condition-based fuzzing operations. Condition-based fuzzing incorporates simulator state information together with logical formulas to determine when to activate and deactivate fuzzing.

The simulation-based testing platform provides an automated approach for exploring the testing space. During its loop, new testing scenarios are generated, executed and their metric values are explored. 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. Importantly, while the genetic algorithm focuses on the most interesting regions in the testing space, which produce the most violations, the coverage of the parameter space of available fuzzing tests is considered also.

We propose a testing DSL in order to describe and constrain 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. Our DSL acts both as a configuration tool and repository for performed experimental results.

When considering coverage, a dimensional reduction strategy is explored, in which the parameter space of fuzzing operations is reduced down to a more tractable size, allowing a reasonable proportion to be explored during fuzzing experiments. This dimensionality reduction can be executed online during evolution to assess the proportion of the space covered, and also executed at the end of the experiment to extract the dimensional positions of the output fuzzing configurations. From this output, we presented a subset selection algorithm that can select configurations for physical testing, supporting a tradeoff between exploitation of the best results and exploration of the parameter space to find potentially widely spaced reality gaps.

We have presented a case study from our KUKA industrial partners for the use case incorporating the TTS simulator, demonstrating these techniques operating from the scenario analysis to the selection of a subset of results for physical testing. In this approach, we have discussed how the platform can interface with EDDIs, presented an example EDDI for ensuring safe operation, and considered how the platform can verify safe EDDI behaviour.

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. IEEE, 2020.

[2] Rob Alexander, Heather Rebecca Hawkins, and Andrew John Rae. Situation coverage ? a coverage criterion for testing autonomous robots. Report, University of York, February 2015.

[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] Alberto Bartoli, Mauro Castelli, and Eric Medvet. Weighted hierarchical grammatical evolution. *IEEE Transactions on Cybernetics*, 50(2):476–488, 2018.

[8] 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.

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

[10] Marcel Boehme, Cristian Cadar, and Abhik Roychoudhury. Fuzzing: Challenges and reflections. *IEEE Softw.*, 38(3):79–86, 2021.

[11] Marcel Böhme. Stads: Software testing as species discovery. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 27(2):1–52, 2018.

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

[13] 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.

[14] 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.

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

[16] Peng Chang and Taşkin Padif. Sim2real2sim: Bridging the gap between simulation and real-world in flexible object manipulation. In *2020 Fourth IEEE International Conference on Robotic Computing (IRC)*, pages 56–62, 2020.

[17] 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.

[18] 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.

[19] 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.

[20] Iván García Daza, Rubén Izquierdo, Luis Miguel Martínez, Ola Benderius, and David Fernández Llorca. Sim-to-real transfer and reality gap modeling in model predictive control for autonomous driving. *Applied Intelligence*, 53(10):12719–12735, oct 2022.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] Tommaso Dreossi, Daniel J. Fremont, Shromona Ghosh, Edward Kim, Hadi Ravanbakhsh, Marcell Vazquez-Chanlatte, and Sanjit A. Seshia. VerifAI: A toolkit for the formal design and analysis of artificial intelligence-based systems. In *31st International Conference on Computer Aided Verification (CAV)*, July 2019.

[27] 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.

[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] Sreenivas Gollapudi and Aneesh Sharma. An axiomatic approach for result diversification. In *Proceedings of the 18th International Conference on World Wide Web*, WWW '09, page 381–390, New York, NY, USA, 2009. Association for Computing Machinery.

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

[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] 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.

[35] Nick Jakobi, Phil Husbands, and Inman Harvey. “noise and the reality gap: The use of simulation in evolutionary robotics”. In *Lecture Notes in Artificial Intelligence*, volume 929, pages 704–720, 01 1995.

[36] Rae Jeong, Jackie Kay, Francesco Romano, Thomas Lampe, Thomas Rothörl, Abbas Abdolmaleki, Tom Erez, Yuval Tassa, and Francesco Nori. Modelling generalized forces with reinforcement learning for sim-to-real transfer. *CoRR*, abs/1910.09471, 2019.

[37] 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.

[38] Jacky Liang, Saumya Saxena, and Oliver Kroemer. Learning active task-oriented exploration policies for bridging the sim-to-real gap. In *Robotics: Science and Systems 2020*, 07 2020.

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

[40] 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.

[41] 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.

[42] Antonio J. Nebro. Nsga-ii variants. =http://jmetal.sourceforge.net/nsgaII.html. Accessed: 2022-06-28.

[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] Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. Fuzzfactory: domain-specific fuzzing with waypoints. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–29, 2019.

[46] Richard F Paige, Dimitrios S Kolovos, and Fiona AC Polack. A tutorial on metamodelling for grammar researchers. *Science of Computer Programming*, 96:396–416, 2014.

[47] 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.

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

[49] SESAME Project Partners. D3.2: Executable scenarios workbench (initial version). Technical report, The Open Group, Jun 2022.

[50] SESAME Project Partners. D4.3: Safety-security co-engineering framework. Technical report, The Open Group, Jun 2022.

[51] SESAME Project Partners. D6.2: Section 3.4.1: Simulation-based testing methodology for EDDIs. Technical report, The Open Group, Jun 2022.

[52] SESAME Project Partners. D6.2: Simulation-based testing methodology for EDDIs. Technical report, The Open Group, Jun 2022.

[53] SESAME Project Partners. D7.1: Runtime safety and security concept – executable digital dependability identity (EDDI) runtime model specification. Technical report, The Open Group, Jun 2022.

[54] SESAME Project Partners. D7.2: Tools for generation of runtime EDDIs. Technical report, The Open Group, Jun 2022.

[55] SESAME Project Partners. D8.8: Management of mrs-based assembly lines use case evaluation - interim version. Technical report, The Open Group, Oct 2022.

[56] SESAME Project Partners. D6.5: Quality assurance of EDDI-enabled MRS using digital twins. Technical report, The Open Group, Jun 2023.

[57] SESAME Project Partners. D6.7: Simulation-based testing methodology for EDDIs. Technical report, The Open Group, Jun 2023.

[58] 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.

[59] 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.

[60] Fabio Ramos, Rafael Possas, and Dieter Fox. Bayessim: Adaptive domain randomization via probabilistic inference for robotics simulators. In Antonio Bicchi, Hadas Kress-Gazit, and Seth Hutchinson, editors, *Robotics: Science and Systems XV, University of Freiburg, Freiburg im Breisgau, Germany, June 22-26, 2019*, 2019.

[61] Fabio Reway, Abdul Hoffmann, Diogo Wachtel, Werner Huber, Alois Knoll, and Eduardo Ribeiro. Test method for measuring the simulation-to-reality gap of camera-based object detection algorithms for autonomous driving. In *2020 IEEE Intelligent Vehicles Symposium (IV)*, pages 1249–1256, 2020.

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

[63] Clément Robert, Jérémie Guiochet, and Hélène Waeselynck. Testing a non-deterministic robot in simulation - how many repeated runs ? In *2020 Fourth IEEE International Conference on Robotic Computing (IRC)*, pages 263–270, 2020.

[64] 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.

[65] 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.

[66] 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.

[67] 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.

[68] 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.

[69] 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.

[70] 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.

[71] TTS S.r.l. Ddd simulation suite. =https://www.ttsnetwork.com/en/simulator/. Accessed: 2023-06-16.

[72] A. Stocco, B. Pulfer, and P. Tonella. Mind the gap! a study on the transferability of virtual versus physical-world testing of autonomous driving systems. *IEEE Transactions on Software Engineering*, 49(04):1928–1940, apr 2023.

[73] SAFEMUV Project Team. Safemuv: Deliverable report d3.3: Safe airframe inspection using multiple uavs. Technical report, University of York, Jan 2022.

[74] SAFEMUV Project Team. Saffemuv: Deliverable report d4.1: Safe airframe inspection using multiple uavs. Technical report, University of Luxembourg, Feb 2022.

[75] 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.

[76] 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.

[77] Ya-Yen Tsai, Hui Xu, Zihan Ding, Chong Zhang, Edward Johns, and Bidan Huang. Droid: Minimizing the reality gap using single-shot human demonstration. *IEEE Robotics and Automation Letters*, PP:1–1, 02 2021.

[78] Eugene Valassakis, Zihan Ding, and Edward Johns. Crossing the gap: A deep dive into zero-shot sim-to-real transfer for dynamics. In *International Conference on Intelligent Robots and Systems (IROS)*, 2020.

[79] 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.

[80] 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.

[81] 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.

[82] 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.

[83] Wenhao Yu, Jie Tan, C. Karen Liu, and Greg Turk. Preparing for the unknown: Learning a universal policy with online system identification. In Nancy M. Amato, Siddhartha S. Srinivasa, Nora Ayanian, and Scott Kuindersma, editors, *Robotics: Science and Systems XIII, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA, July 12-16, 2017*, 2017.

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

[85] 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.

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

[87] Shaojun Zhu, Andrew Kimmel, Kostas E. Bekris, and Abdeslam Boularias. Fast model identification via physics engines for data-efficient policy search. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, IJCAI'18, page 3249–3256. AAAI Press, 2018.