Project Number 101017258

D6.7 Tools for Automated Quality Assurance of EDDI-Supported MRS (Final Version)

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**

Every effort has been made to ensure that all statements and information contained herein are accurate, however the SESAME Project Partners accept no liability for any error or omission in the same.

© 2023 Copyright in this document remains vested in the SESAME Project Partners.

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	First draft	2 June 2023
0.3	Completed draft	20 June 2023
0.8	Final draft after review	29 June 2023
1.0	Completed QA version	7 July 2023

Table of Contents

1 Introduction **1**

2 Simulation-Based Testing **2**

2.1 Scope Of Simulation-Based Testing 2
2.2 Architecture of Integrated Testing 2
2.2.1 Testing Platform Interfaces 3
2.2.2 DSL Definition for MRS 4
2.2.3 Simulator-Specific Interfaces 6
2.2.4 EDDI Runtime Interfacing. 9
2.2.5 New Package Structure and Custom Performance Metrics 9
2.2.6 Test Generation and GA Execution 9
2.2.7 Simulation Results and Traces 9
2.2.8 Subset Selection 10
2.3 User Guide 10
2.3.1 Setting Up The Environment - Required Libraries 10
2.3.2 Installation Instructions 11
2.3.3 User Guide 11
2.4 Simulation-Based Testing Execution Example 11

3 Hardening and Repairing of Deep Learning Components **18**

3.1 Scope Of Hardening and Repairing Recommendations 18
3.1.1 GENERATIVEREPAIR Tool 18
3.2 GENERATIVEREPAIR Architecture 19
3.3 User Guide 20
3.3.1 Setting Up the Environment: Required Libraries 23
3.3.2 Running the GENERATIVEREPAIR Tool 23
3.4 GENERATIVEREPAIR Execution Example 24

4 Conclusion **29**

List of Figures

1 Multi-Stage Quality Assurance Methodology developed for Task 6.5 3
2 Package diagram showing dependencies between packages 5
3 Package diagram showing dependencies between packages 5
4 Simulation interfacing for remapping in ROS. Circles illustrate the simulator components, the cut rectangles illustrate the simulator variables, and the pentagon the added simulation-based testing infrastructure 7
5 KUKA interfacing with the physical simulation 8
6 Evolution results, both final and intermediate following GA evaluation (stored in the CampaignResultSets class) 10
7 Generation steps for the SESAME simulation-based framework 12
8 Simulation-based testing methodology 13
9 Testing Space Model for KUKA/TTS example 13
10 Code generation by activating SESAME wizard 14
11 The metric implementation for collision occurrence 15
12 Example result sets for example experiment 16
13 Extracting the model contents from the result set to process 17
14 SubsetSelection notebook which allows setting parameters, executing and visualising the subset selection, and listing the chosen configurations 17
15 Overview of the Hardening and Repairing Methodology @DesignTime 19
16 Class diagram for GenerativeRepair showing dependencies between components. 22
17 Parameters for configuring GenerativeRepair 25
18 Examples of the repairing outputs. A set of synthetic images generated using Random+Inpainting fuzzer. All these images are augmented using the same input seed. 26
19 Shell command to execute “*repairing.py*” script- 1 27
20 Shell command to execute “*repairing.py*” script- 2 27
21 Shell command to execute “*repairing.py*” script- 3 28

Executive Summary

This document supports the prototype tools developed in Tasks 6.3 – 6.5 and provides information on the tools developed. The prototype tools implement the requirements, principles, and architecture described in deliverables D6.4 and D6.6. To this end, D6.7 reports on the interfaces and connectivities of the tools, and how they fit into the associated methodologies.

The simulation-based testing section summarises the simulation-based testing process, and provides information on the structure of the code. We describe the simulation-based testing toolkit, presenting the integrated methodology, interfaces of various components, and a usage example.

Similarly, the section on repairing deep learning components, summarises the GenerativeRepair and describes the architecture of the developed tool. Furthermore, a usage example using the Grape Leaves dataset is provided.

The document provides links to information for installing the developed prototype tools and information on using these tools, complementing the information provided on the tools Github repositories.

1 Introduction

This deliverable provides the final version of the quality assurance toolset, including quality assurance tools for data-driven learning components, simulation-based testing tools for EDDIs, EDDI debugging and hardening tools, and a digital twins technology. This toolset has been developed in the context of Tasks 6.1-6.6 and the technological and scientific innovations have been presented in the other WP6 deliverables.

The simulation-based testing toolset is described in Section 2, which implements the methodologies and architectures (presented in Deliverable D6.6 [14]) for integrated testing, beginning with simulation-based testing and managing the transition between simulation-based testing and lab experimentation. Following a specification of the scenario requirements, custom performance metrics and an EDDI, the chosen scenario can be explored with simulation-based testing. We use 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 procedure that dynamically synthesises and executes experiments, performing dynamically generated test campaigns, quantified with scenario-specific performance metrics. Following this, we present a tool to select a subset of the output configurations discovered in evolution for physical testing.

The quality assurance tool for Data-Driven and Learning components is described in Section 3, which implements the methodologies and architectures presented in deliverable D6.4 [18] (1) GENERATIVEFUZZER, a coverage-guided fuzzing technique for test case generation; and (2) SAFETYREPAIR, a continual learning technique for repairing Deep Learning systems. Both tools are implemented and integrated into a self-contained fuzz testing and repairing framework named GENERATIVEREPAIR.

GENERATIVEREPAIR is an extensible framework, implemented as a set of Python components. These components allow test engineers and system integrators to customise the testing and repairing operations and pass in parameters from the command line. Our quality assurance tool improves the robustness of Data-Driven and Learning components, i.e., Deep Learning-based systems, through semantic data augmentation methods that represent natural environmental patterns. In the following, we provide a comprehensive presentation of our tools and give step-by-step usage examples.

This document is structured as follows:

- Section 2 describes the simulation-based testing toolkit, presenting the integrated methodology, the interfaces of the testing toolkit, installation requirements, and a usage example
- Section 3 describes the hardening and repairing framework, its architecture, a detailed user guide, installation requirements, and a full execution example.
- Section 4 concludes this deliverable.

2 Simulation-Based Testing

2.1 Scope Of Simulation-Based Testing

This section presents an overview of the tools for the simulation-based testing components of SESAME, and a brief description of the SESAME testing methodology together with a description of the interfaces provided and usage information. The information provided in this section complements deliverable D6.6 [14].

Simulation-based testing enables investigating the capacity of a multi-robot system (MRS) to operate dependably using a virtual environment [2, 3, 7, 19, 21]. Simulated MRS systems can be subject to reality gaps, being significantly different in terms of physical features, modelling assumptions and response to transient faults or interference [4, 20].

The requirements for the simulation-based testing are:

- Allow users to characterise the performance of the MRS in a flexible way, according to scenario-specific metrics
- Inform the automated discovery of faults according to an algorithm that permits incremental improvement of discovered configurations
- Devise a structured process to transition from simulation-based testing to physical testing

These requirements have been implemented in a toolset for simulation-based testing, which consists of a number of Eclipse-based projects and associated analysis tools that discover faults by simulation and select configurations for physical testing. The outputs of the toolset are result sets showing the most useful detected faults according to the defined performance requirements, and configurations that can be passed through to physical testing in order to determine any reality gaps between the lab system and performance testing.

2.2 Architecture of Integrated Testing

The overall methodology for the transition from simulation to physical testing is illustrated in Figure 1, which comprises six distinct stages. The human symbol indicates those stages which involve the use of human assessment or intervention (e.g., 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). The underlying conceptual and algorithmic details of these stages are detailed in the appropriate sections in our deliverable D6.6 [14]. The technical implementation and interfacing for each of the stages of the methodology are described below:

Step 1 **Scenario Definition**: Receives information from the scenario requirements, the EDDI specification and ExSce. Provides a concrete set of quantitative metrics for performance monitoring in terms of metric definitions, which will be fed into Steps 2 and 3. The fuzzing operations to be used in the testing of the scenario will be selected here, and any operations that require custom operations will be setup in Step 3

Step 2 **Preparatory testing**: Uses gathered data from the system and the scenario-specific interface for the simulator. During this step, the testing platform is used in passive monitoring mode. It will be necessary to perform Step 3.3 on metric definition in order to setup the performance metrics. (If using the "Known Fuzzing Tests" of Step 2, (documented in D6.6 [14]) it will be necessary to define these initial fuzzing operations in order to test the system).

Step 3 **Simulation-based testing**: The evolutionary algorithms documented in D6.6 [14] are used to evolve a population of tests. This uses the interfaces described in Section 2.2.1, namely the definition of the performance metrics, the GA specification, and a concrete specification of the MRS (defined in Section 2.2.2)

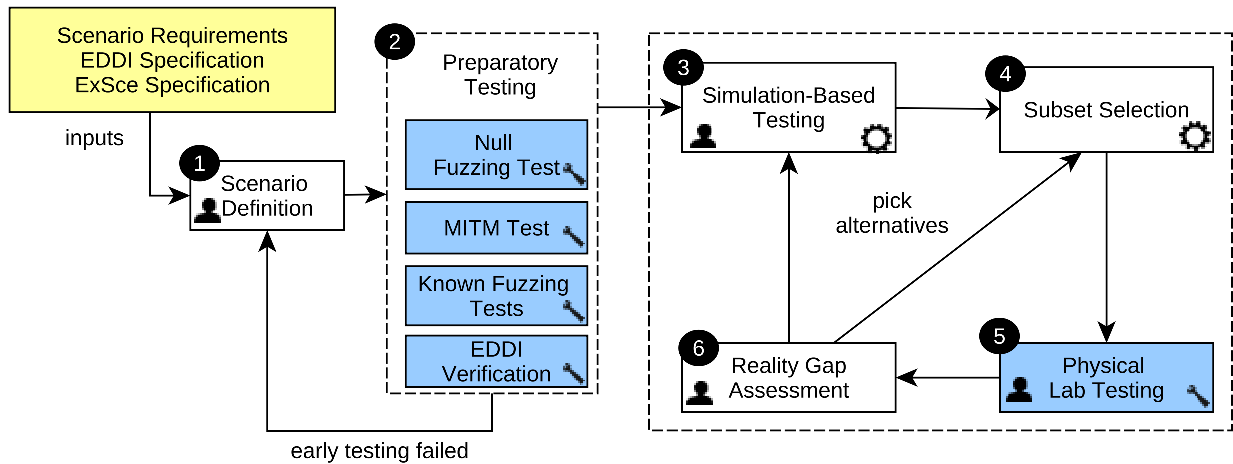


Figure 1: Multi-Stage Quality Assurance Methodology developed for Task 6.5

Step 4 **Subset Selection**: Provides the selection of the subset of scenarios explored under Step 3 for physical testing. Uses the results stored in the output model file from the previous stage, with conversion by Java into a CSV file. Subset selection is implemented in Python and provided as a Jupyter notebook, which reads the CSV file and provides an interface to set the optimisation parameters (such as λ and maximum cost). The notebook also provides visualisation of the chosen configurations in parameter space. Further details of this process are given in Section 2.2.8

Step 5 **Physical/Lab Testing**: The testing platform will be used with the appropriate scenario-specific interface to connect to the physical scenario. The testing platform will apply fuzzing and collect physical performance metrics, using the set of configurations selected in Step 4. The output results of physical testing (final metric values and metric traces) will be recorded by the platform, stored in the model in order to support the comparison between simulation and physical testing in Step 6

Step 6 **Reality Gap Assessment**: Human assessment of metric results in the output model (containing the experimental results). Recorded performance metrics values and metric traces generated in Steps 3 and 5 may also be used in low-level analysis of reality gaps. (The low-level analysis of reality gaps here refers to analysing the time series of simulator variable or metric values in an attempt to discover the causes of discrepancies between simulation and reality, in a similar manner to [4])

2.2.1 Testing Platform Interfaces

This section presents an overview of the implementation of the testing platform, its interfaces available for connection to external components, and the structure of the DSLs used in interfacing to define the simulation scenarios. The testing DSL is specified in D6.6 [14]. The code for the testing platform is available as open-source project, and is currently available at `https://github.com/sesame-project/simulationBasedTesting`. The platform will be used in simulation testing mode in Steps 2 and 3, and in physical testing interfacing in Step 5.

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 incorporating the Maven build tool [6] are used to recompile code components dynamically generated during the execution of the experiments. Kafka provides 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 interdependent Eclipse projects. All projects (except for the generator project) are Maven based, allowing their dependencies to be automatically downloaded. A package diagram showing the dependencies between these projects is presented in Figure 2. The purpose and structure of some of the important simulator-independent system projects is summarised below:

uk.ac.york.sesame.testing.architecture: This project contains the *ISimulator* interface, which the users must implement in order to interconnect an MRS simulator 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 the associated serialisation and deserialisation code.

uk.ac.york.sesame.testing.dsl: We store the models of the simulation-based testing DSL involved in the project. The current version of the Testing DSL described in deliverable D6.6 [14] is stored in the Emfatic representation (under TestingMM.emf). The MRS DSL is contained in the package MRS under this file. 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 EGL generators for TTS and ROS middlewares for test execution, and generators for the implemented fuzzing operations

uk.york.sesame.testing.evolutionary This package provides support for evolutionary experiments using the JMetal framework [13], 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 [1]. 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 to be processed in the subset selection phase described in Section 2.2.8.

2.2.2 DSL Definition for MRS

The information in the ExSce is used to define an MRS model that serves as an input to the testing platform. A UML diagram of the MRS model is provided in Figure 3. The information in this is used to control the sub-

[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

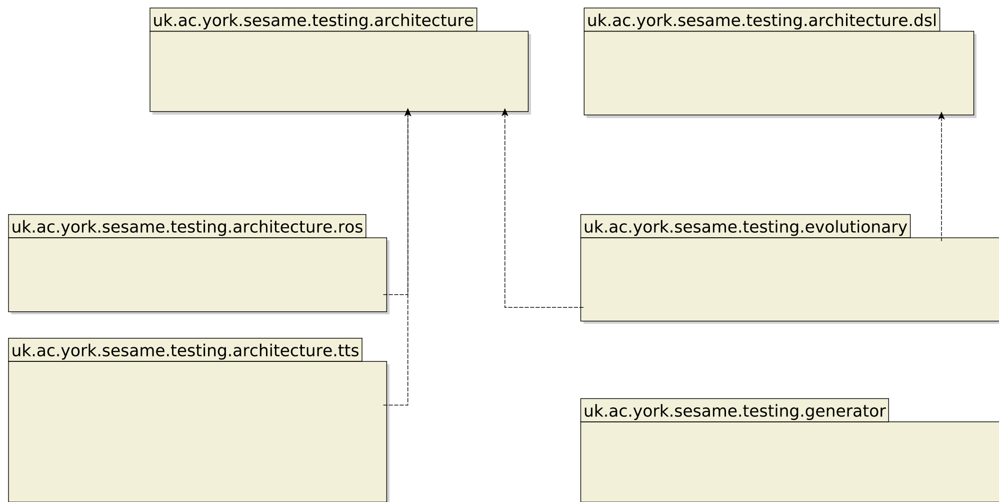


Figure 2: Package diagram showing dependencies between packages

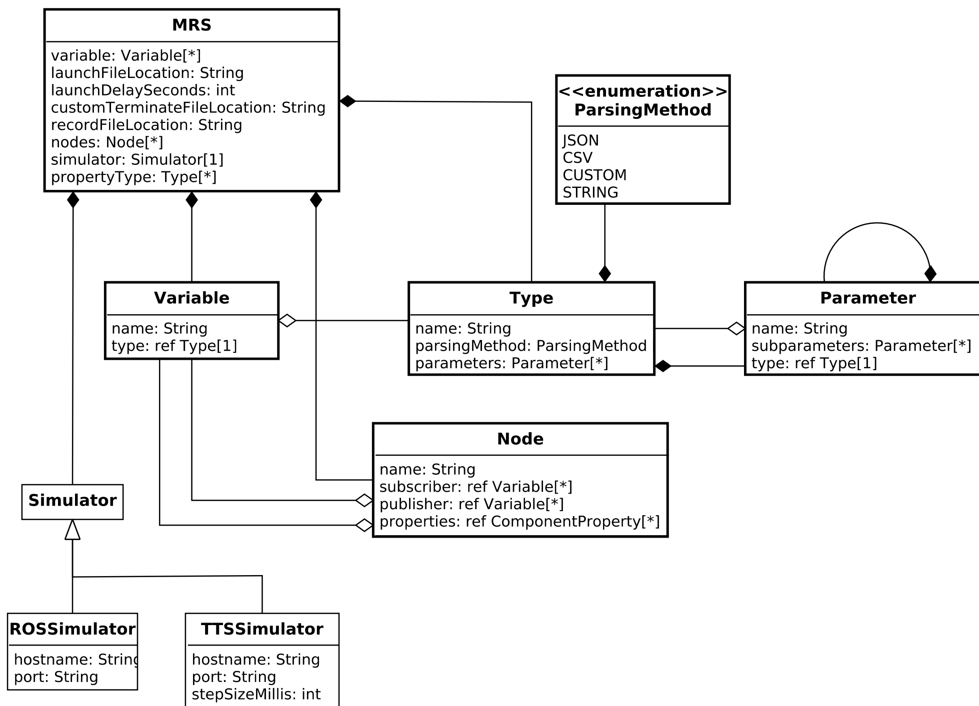


Figure 3: Package diagram showing dependencies between packages

scriptions to simulator variables, by specifying information about their data types and the logical structure of the system in terms of transmitting and receiving nodes. The selected type of information and parsing method also alters the internal operation of several fuzzing operations; for example, selecting the JSON ParsingMethod when using ROS allows the use of structured variables, while within raw variables, the STRING parsing method should be used. These MRS variables are referenced when specifying the testing model as defined in D6.6 [14] (e.g., fuzzing operations are selected to operate upon a particular variable). Further information about the MRS model structure and examples for KUKA- and ROS-based projects are presented in the User Guide online at: `https://github.com/sesame-project/simulationBasedTesting/blob/main/documentation/userguide.md`.

2.2.3 Simulator-Specific Interfaces

The SESAME testing platform provides an extensible interface for implementing new simulator interfaces, allowing the simulation-based testing platform to be expanded to interface with other MRS simulators. The architecture presents a Java interface, *ISimulator.java* in package `uk.ac.york.sesame.testing.architecture`, which users must implement to connect their simulator with the provided infrastructure. Key methods related to MRS communications that must implement this interface are presented below:

public Object connect(ConnectionProperties params) This method provides the code for connecting the middleware to the MRS simulator (e.g., for ROS, making a connection to the simulator via *rosbridge*). The supplied parameters are used to configure the connection, for example, supplying the hostname and port for the MRS connection. These parameters are configured in the MRS model definition in Section 2.2.2 (Figure 3).

public void consumeFromVariable(String varName, String varType, Boolean publishToKafka, String kafkaTopic,. . .) The implementation of this method provides a mechanism for consuming the variables of the robotic system by the testing platform. The first parameter is the name of the variable in the robotic system that we need to consume, the second is its type. The third is used to define if incoming messages should be redirected to Kafka (it should always be true in practical use, as Kafka is used internally for message processing). The last parameter declares the name of the Kafka topic to receive the message.

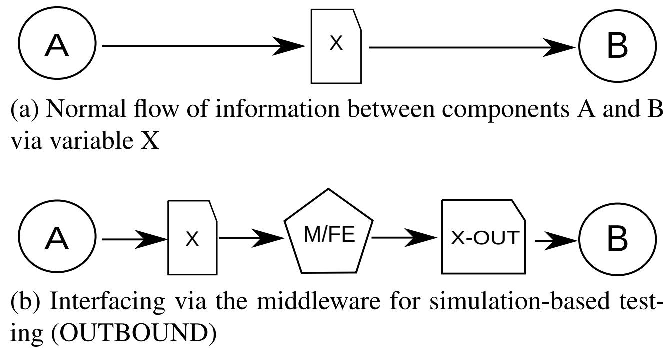
public void publishToVariable(String varName, String varType, String message) The implementation of this method provides the mechanism to publish a message back to a variable of the robotic system. The parameters specify the variable name, its type and the message itself (as a String).

public void updateTime() In the testing platform, the only concept of time used is always the MRS simulator time. Wall-clock time is only used in debugging and internal monitoring of latency during communication. This allows the MRS simulation to potentially run at a higher speed than wall-clock time, if permitted by performance constraints and configuration of the underlying MRS simulator. By implementing this method, users provide the mechanism to update the timestamps in the architecture by collecting them from the underlying mechanism of the MRS (e.g., for ROS, by monitoring the `/clock` topic).

public void stepSimulator() This method is available on an experiment branch of the repository "stepping-tts-interface" for increasing determinism during repeated runs of the same configuration. It supports an alternative approach in which simulation time can be advanced under testing platform control, in order to avoid timing issues potentially caused by the MRS simulator operating independently of the testing platform. The step size is set by providing a $STEP_SIZE$ parameter during connection setup. Whenever *stepSimulator()* is called, the simulation will advance by the given timestep. Further details are provided in the consideration of the TTS simulator interface.

Runtime Simulator Interfacing On startup, the middleware configured for a particular test will make subscriptions to the simulator (e.g., to obtain the topics selected for fuzzing and perform the man-in-the-middle (MITM) approach for variable modification, and any information relevant to monitoring performance metrics to observe safety violations). This information is placed into an Apache Kafka queue named IN, and stream transformations are applied to achieve the fuzzing modifications. The middleware also generates code to receive from a Kafka OUTPUT queue, which transmits potentially modified messages back to the MRS simulator.

ROS interface The ROS simulator models the system components as a graph of interacting nodes. Nodes can publish and subscribe to topics in order to communicate with other robotic components. In the ROS implementation, the term topic is used to denote a simulator variable as described in our DSL model. When a test runner aims to manipulate the MRS internal state, the message flow between source and destination must be intercepted. For example, when a subscription is made to variables published by simulator component A, the simulation configuration must be modified to ensure that the original destination component B only listens to alternative variables with OUT appended to their names. The middleware MITM mechanism will listen to the original variable X, transform the messages according to the given fuzzing operation(s), and republish



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

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

Figure 4: Simulation interfacing for remapping in ROS. Circles illustrate the simulator components, the cut rectangles illustrate the simulator variables, and the pentagon the added simulation-based testing infrastructure

the modified values as X-OUT. This is illustrated in Figure 4b. In order to support this remapping, system testers must provide modified ROS launch scripts ahead of MRS execution, which have been altered in order to contain the remapped names.

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

The package `uk.ac.york.sesame.testing.architecture.ros` contains an interface to ROS/Gazebo simulations (ROSSimulator), which implements ISimulator. Its simulation interface uses *jrosbridge*, which provides a Java interface using the standard package ROSbridge upon the MRS side. Although this ROS simulation interface via *jrosbridge* currently supports ROS1, *jrosbridge* has been tested internally using ROS2, with successful receipt of messages[8].

KUKA Interfacing. The TTS simulator-side interface for simulation-based testing is implemented as a gRPC protocol server [8] to which our simulation-based testing middleware can connect. The gRPC server-side interface hides the internal implementation of TTS simulator state, providing a message abstraction, so the simulator side can receive standardised ROS-like messages. A protobuf protocol definition (SimLogAPI.proto) [9] is provided in our SESAME repositories which can be converted into a Java API by the *protobuf-maven-plugin*. This allows the protocol Java API to be updated dynamically whenever the protocol is updated, which supports convenient development, responding to changes made and ensuring the Java-side interface is consistent with the protocol used by TTS.

In our KUKA use case, a heterogeneous topology is employed (Figure 5) in which the TTS simulator and other components such as either Simit or PLC (programmable logic controller) are interconnected and communicate at runtime. During simulation, Simit is used to implement the scenario logic, controlling the robotic joints and actuators and reading sensor status to trigger actions. During physical testing, the PLC controls the real robots in the scenario and handles responses to sensor events. Even in the physical implementation, the TTS simulator is still required, since it may be necessary to perform computations for the performance metrics (such as detection of collision events by 3D intersection), that cannot be assessed by physical sensors.

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

[8] Note that as this is still a development version some issues with ROS2 interfacing may exist that could require changes for message formatting for Stamped messages in order to successfully interface with ROS2; see `https://github.com/rctoris/jrosbridge/issues/33`

[9] `https://github.com/sesame-project/simulationBasedTesting/blob/main/uk.ac.york.sesame.testing.architecture.tts/src/main/proto/SimlogAPI.proto`

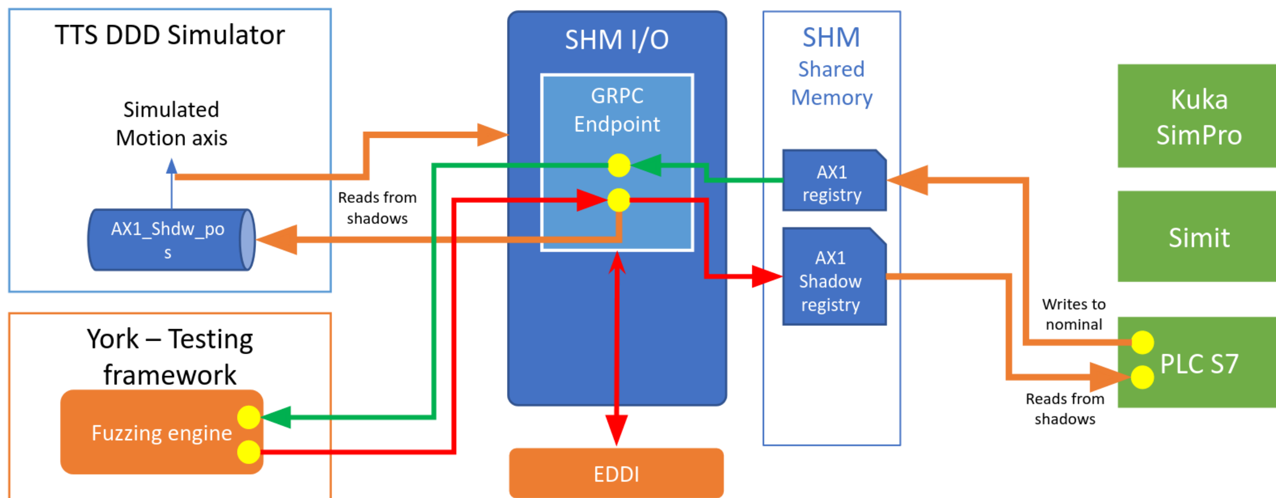


Figure 5: KUKA interfacing with the physical simulation

In order to support the heterogeneous topology without requiring a distributed architecture (which would present challenges, potentially, error-prone and unstable locking and update protocols), a single shared memory component has been developed. This component acts as a central point for state storage, and a single point of connection for both the SESAME simulation-based testing platform and the EDDI.

On the testing framework side, a class TTSSimulator has been developed, as an implementation of the ISimulator interface. Its *connect* method makes a connection to the simulator over a gRPC channel, creating blocking and non-blocking gRPC-Java stubs. The *consumeFromVariable* function creates a custom ROSObserver, which subscribes and is notified when messages arrive from the channel. This ROSObserver connects to the Apache Kafka internal input stream for the middleware, and pushes messages received there into this stream for processing. The *publishToTopic* function creates a new PubRequest, and sets it to contain the given message, before handing it off to be transmitted by the non-blocking stub.

The code for the TTS interface is available in `uk.ac.york.sesame.testing.architecture.tts`, with the majority of code in this package being auto-generated from the gRPC protocol definition on installation. The TTS interface is currently being extended with a new stepping mechanism (in the “stepping-tts-interface” Github branch) which can provide increased determinism. When this stepping interface is enabled, configuration options should be set to activate the TTS simulation in a paused state, and simulation time will only advance under the control of the testing platform. The middleware event processing loop ensures that time is only advanced in discrete steps when the Kakfa OUT queue is empty, which ensures that all messages sent over the man-in-the-middle (MITM) and due to be handled by the simulator have been processed before advancing time. This avoids potential non-determinism caused by timing offsets between the testing platform and MRS code execution (such as joint updates at critical points in path planning).

Other Simulator Interfaces. In simulators that provide external interfaces (such as over TCP/IP or message bus protocols, or a plugin API) that allows runtime message monitoring/modification, it may be possible to include standardised interfacing component/plugins and modify the simulation configuration in order to deliver messages over this MITM. If standardised interfacing protocols are not supported but source code is available, a custom connection endpoint can be implemented in the source code to which the testing platform can connect (the latter is an approach used with the MOOS-IvP simulator, adding an ATLASDBInterface component in our previous ATLAS project [9]). Attention must be paid to hardcoded variable names used throughout the codebase, since it is important to ensure that recipient components use the fuzzed value received over the MITM, not the original one. Regardless of the approach selected, it is then necessary to provide an implementation of the ISimulator interface. This will establish a connection to the simulator logical interface and define functions to consume and publish messages over the defined custom protocol.

2.2.4 EDDI Runtime Interfacing.

The EDDI acts as a runtime monitor above the safety concept, considering whether the MRS and associated sensors can still guarantee that safety concept assumptions hold. The EDDI runs as a separate process and is interconnected using the same mechanism of the underlying MRS. For example, in ROS, the EDDI would be implemented as a ROS node, subscribing to topics for its input variables, and publishing back its decisions on safety guarantees. In the KUKA case study, the EDDI is interconnected directly over gRPC to the shared-memory interfacing component (Figure 5), subscribing to selected input variables and publishing back its decision as a Boolean $EDDI_SAFE$ variable. This common interface allows the testing platform to fuzz the inputs to the EDDI, examining how its decision(s) may be impacted or distorted due to the fuzzing information provided. The simulation-based testing platform can also be used to assess 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.

2.2.5 New Package Structure and Custom 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, users provide an instance of the SESAME Testing DSL, specifying the structure of the testing space, as defined in D6.6 [14]. 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, which provides the interface to specify the performance metric.

2.2.6 Test Generation and GA Execution

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 [5, 13] 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. Coverage boosting can be activated by selecting a specific parameter in the Testing DSL under the NSGAIICoverage-WithCells, *useMutationEnhancingCoverage*, that specifies a custom mutation operation.

2.2.7 Simulation Results and Traces

During the simulation-based testing with GA evolution, the output results (both the full population and non-dominated results on the Pareto front) are stored under the campaign being executed. Instances of the CampaignResultSet class of the DSL (illustrated in the Methodology section of D6.6 [14]) represent either the final outcome of a particular campaign, or partial intermediate results obtained during its execution. This allows the progress of the evolution to be examined as it is ongoing, as well as on completion. An enumeration (ResultSetStatus) which is set to either FINAL or INTERMEDIATE, determines the status of a result set. For example, in an evolutionary experiment, the set of Tests for a final result set, containing the name string NON-DOM would contain the output Pareto front obtained during an experiment. These results can be browsed and examined using the Exceed editor for the output model (Figure 6).

Campaign Result Set NONDOM-20_06_2023_17_01_16-intermediate-140
Campaign Result Set FULL-20_06_2023_17_16_31-intermediate-150
Campaign Result Set NONDOM-20_06_2023_17_16_31-intermediate-150
Campaign Result Set FULL-20_06_2023_17_31_53-intermediate-160
Campaign Result Set NONDOM-20_06_2023_17_31_53-intermediate-160
Campaign Result Set FULL-20_06_2023_17_47_18-intermediate-170
Campaign Result Set NONDOM-20_06_2023_17_47_18-intermediate-170
Campaign Result Set FULL-20_06_2023_18_02_50-intermediate-180
Campaign Result Set NONDOM-20_06_2023_18_02_50-intermediate-180
Campaign Result Set FULL-20_06_2023_18_18_33-intermediate-190
Campaign Result Set NONDOM-20_06_2023_18_18_33-intermediate-190
Campaign Result Set FULL-20_06_2023_18_34_24-intermediate-200
Campaign Result Set NONDOM-20_06_2023_18_34_24-intermediate-200
Campaign Result Set FULL-20_06_2023_18_50_18-intermediate-210
Campaign Result Set NONDOM-20_06_2023_18_50_18-intermediate-210
Campaign Result Set FULL-20_06_2023_19_06_15-intermediate-220
Campaign Result Set NONDOM-20_06_2023_19_06_15-intermediate-220

Figure 6: Evolution results, both final and intermediate following GA evaluation (stored in the CampaignResultSets class)

Result sets contain references to particular tests that comprise the results. Each test contains metric instances that give the final values for the metrics defined in that testing campaign. The Exceed editor allows the output metric results to be inspected. For additional convenience, an EGL script (*files/resultsAnalysis/resultsAnalysis.egl*) is provided under the package `uk.ac.york.sesame.testing.generator` which can analyse all the tests and produce a summary of results to the console. In addition to the final results, it may also be useful to consider the time series of results, which may be useful, for example, when debugging or inspecting non-determinism, or isolating reality gaps from these low-level time-series properties. The DSL definition allows a FileResult to be added for a particular metric, which will then be automatically logged with the time and metric value upon every update during simulation.

2.2.8 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 (available at *<GITHUB_ROOT>/notebooks/phytesting/SubsetSelection.ipynb*) can be loaded which reads this information and implements the subset selection algorithm described in D6.6 [14]. This allows interactively setting parameters such as λ and examining/visualising configurations selected for physical execution.

2.3 User Guide

2.3.1 Setting Up The Environment - Required Libraries

Dependencies for Linux

- Tested under Ubuntu Linux 18.04 (although later versions should work too)
- Apache Kafka / Zookeeper
- ROS installation (tested with ROS Melodic) - if using ROS interface - with rosbridge

Dependencies for Windows

- Tested under Windows 10 - (It may be possible to run on Windows 7, although Windows 10 was used for internal testing)
- Install Docker Desktop - since Kafka uses Docker on Windows
- Apache Kafka, Docker, JDK8 and JDK11
- Administrator access is needed
- Virtualisation enabled in the BIOS (may be defined as "Hyper-V")

2.3.2 Installation Instructions

The installation instructions for Linux are located at the following location: `https://github.com/sesame-project/simulationBasedTesting/blob/main/documentation/INSTALL-linux.md`

The installation instructions for Windows are located at the following location: `https://github.com/sesame-project/simulationBasedTesting/blob/windows/README-windows.md`

2.3.3 User Guide

The user guide is available at: `https://github.com/sesame-project/simulationBasedTesting/blob/main/documentation/userguide.md`

2.4 Simulation-Based Testing Execution Example

Creating a New Project
In order to use the simulation-based testing platform, firstly, the user should load Eclipse and then invoke a new Eclipse Application, by right-clicking upon the newly imported project `uk.ac.york.sesame.testing.generator` and selecting "Run As" / "Eclipse Application". This will launch a fresh Eclipse instance under which the SESAME automated code generation plugins are available (Figure 7a) [10]. There may be a delay on the first invocation of this.

Create a new Java project - here, we use **TTSTestProject**. Create a folder "models" in it. When creating the project, turn off "Create module-info.java" at the bottom of the project dialog.

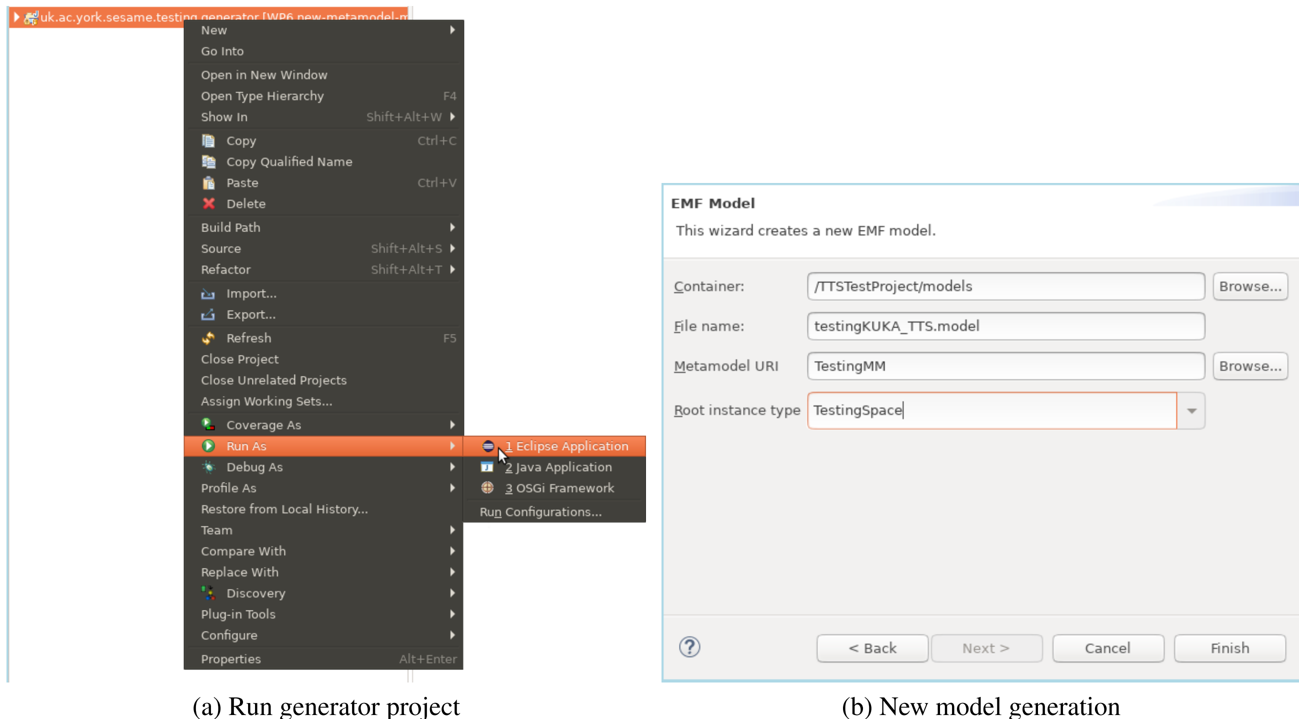
In order to generate a model for the first time in a newly created project, it is necessary to register the metamodels. This can be done by activating the early stage of our wizard, by right-clicking on "SESAME" / "Generate SESAME Code", as shown in Figure 10a. Then, click Cancel on the dialog box that appears.

Next, create an instance of the testing metamodel. To do this, right click on the folder "models" and select "New" / "Other" / "Epsilon" / "EMF Model" and set up the parameters as shown in Figure 7b. Use "Browse" to find "TestingMM" as the metamodel URI. The model filename can be chosen to fit the scenario that the users are setting up.

Now users can complete the steps specified in the SESAME simulation-based testing methodology. When the model is completed, or when there are any changes to the experiments, right click on the project, and select "SESAME" / "Generate SESAME Code" to regenerate code for metric templates and experiment runners based on it (Figure 10).

After the code generation is performed for the first time, the project structure must be changed to a Maven project. Users should right-click on the project, select "Configure" / "Convert to Maven Project". Then the user should right-click and select "Maven" / "Update Project" to ensure all the dependencies are updated.

[10] If there is a problem flagged regarding "javax.xml.bind", please ignore it



(a) Run generator project

(b) New model generation

Figure 7: Generation steps for the SESAME simulation-based framework

An example for the next stage is now given, for setting up an instantiation of the model model for the KUKA case study.

Testing Methodology Example For KUKA
We apply the methodology presented in Figure 8 (for more details, see "Simulation-Based Testing Process" in D6.6 [14]).

Step 3.1: Having analysed the case study requirements, participating robots and the necessary fuzzing operations, users will determine their performance metrics and the fuzzing operations that would like to use. Within the wider integrated methodology of Figure 1, users should provide an instantiation of the SESAME Testing DSL, specifying the structure of the testing space for the example case study. The DSL metamodel is specified in our deliverable D6.6 [14]. The Exceed editor provides a convenient visual editor in order to assist the configuration of the testing process. An example for testing the KUKA example case study is presented in Figure 9. This model includes metrics for quantifying the requirements for testing the particular robots, condition metrics for defining the triggers for condition-based fuzzing and the selection potential fuzzing operations upon different robots. Further details are presented in the case study section of D6.6[14]. An example model file (including later execution results) is presented here [11].

Step 3.2: Code generation can be used to generate metric templates automatically, within the newly generated project under the child Eclipse instance. The testing platform 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" (Figure 10a). The plugin provides an interface option to select the file containing the user's populated model and associated settings (Figure 10b). The annotations in red upon the screenshot show the values selected for the text boxes. Here we choose the model file and the locations of other items for the project:

[11] https://github.com/sesame-project/simulationBasedTesting/blob/main/runtime-EclipseApplication/TTSTestProject/models/example/testingTTS_Kuka_phytesting_coverageGA_example.model

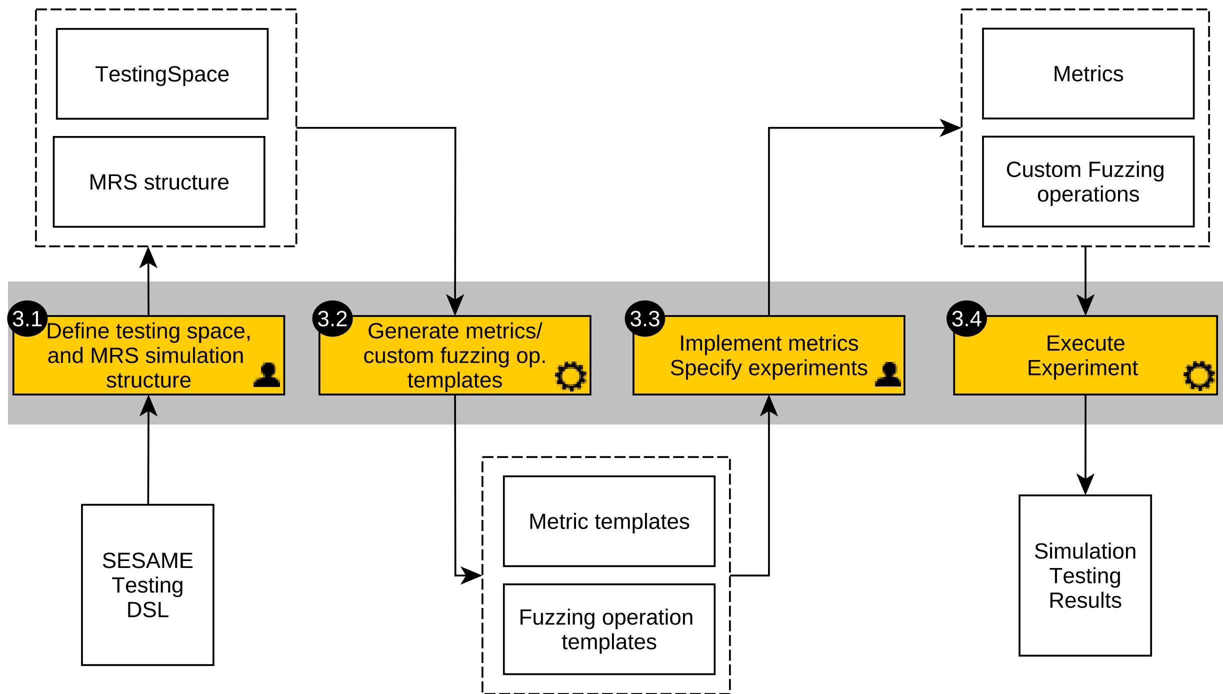


Figure 8: Simulation-based testing methodology

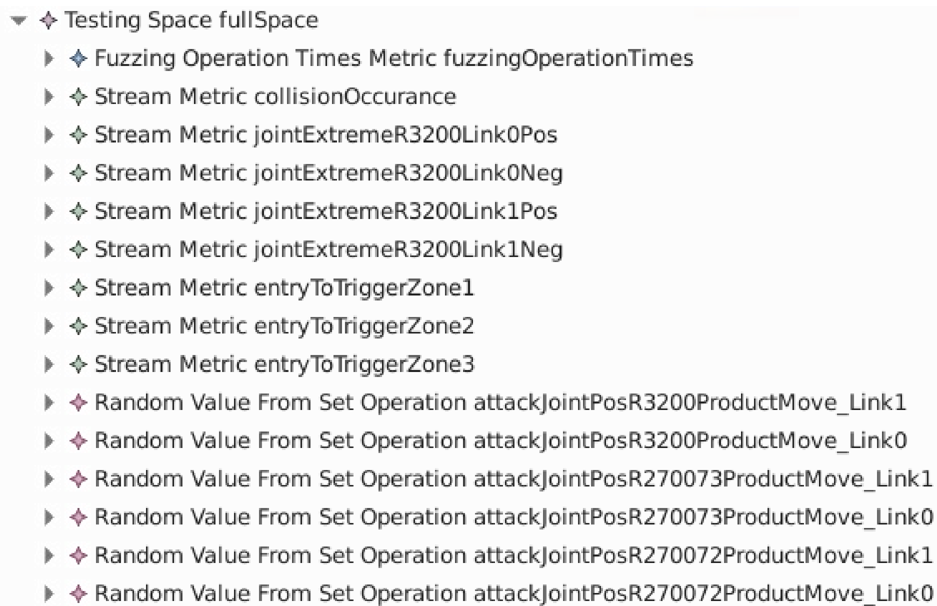
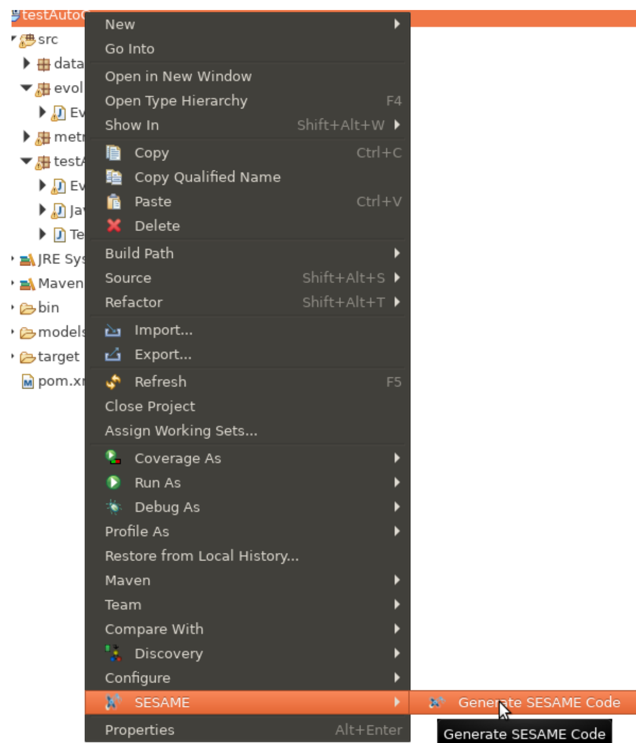


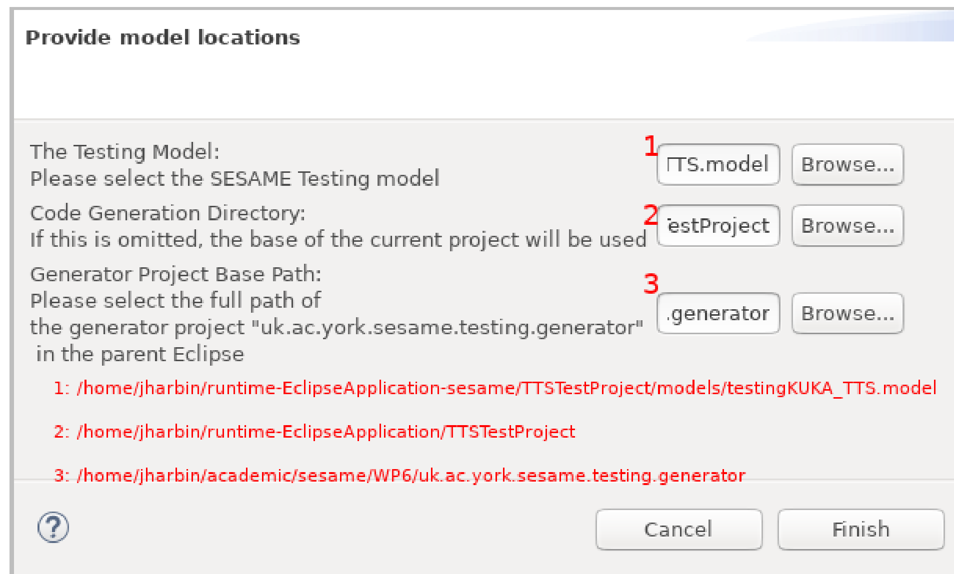
Figure 9: Testing Space Model for KUKA/TTS example

When this button is clicked, metric templates will be generated in the package `metrics.generated`. Experiment runners will also be generated based upon the experiment name defined for the TestCampaigns in the model, e.g., 'ExptRunner_name.java'.

Step 3.3: The next step involves the user specifying scenario-specific performance metrics for the metrics generated in the model, in order to quantify violations of mission requirements. In order to implement these metrics, the user first needs to copy these classes from `metrics.generated` into a new package `metrics.custom`. Then, it is necessary to implement the needed platform-specific metrics as Java code.



(a) SESAME Wizard activation



(b) SESAME Wizard model file selection

Figure 10: Code generation by activating SESAME wizard

Figure 11 presents a fragment of the implementation of the $collisionOccurrence$ metric used to quantify violations by tracking the number of intervals in which collisions of the gripper safety zone with any safety zone. The metric operates as follows:

If the region surrounding the robot gripper (green sphere) collides with these regions, the collision detection logic will trigger a safety zone message which will be sent to the testing platform via the TTSSimulator custom API over gRPC. As an inbound simulator event, these will then trigger the processElement1 method of the metric. If the message inbound topic is a safety zone message of sufficient depth, then the $violationCount$

```
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 11: The metric implementation for collision occurrence

variable will be incremented. This value is emitted as the output value. The final $violationCount$ value will be logged as the output of the metric.

Step 3.4: The user should create an Eclipse Run Configuration for the class **ExptRunner_name.java** for the name of the experiment they would like to execute, and invoke this Run Configuration in order to run the experiment. This runner will be configured with the parameters chosen from the Testing DSL. If any parameters of the experiment in the model are changed, users should rerun "Generate SESAME Code" in Step 3.2, to ensure this experiment runner is updated.

The experiment runner will generate tests according to the strategy specified for the experiement's test campaign. Its TestGenerationApproach selection allows the user to specify the parameters for an experiment by setting one of several subclasses. For example, including **NSGAEvolutionaryAlgorithm** allows an evolutionary experiment with the NSGA-II algorithm [5], and contains specific parameters relevant to this approach, e.g., the number of iterations and the population size. 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 to be 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. Each test is evaluated by first dynamically generating a specialised test runner which acts as a middleware interfacing with the low-level MRS simulation and modifying its internal messages, using any custom-supplied metric definitions provided in Step 3.3 to quantify the impact of the fuzzing test. 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.

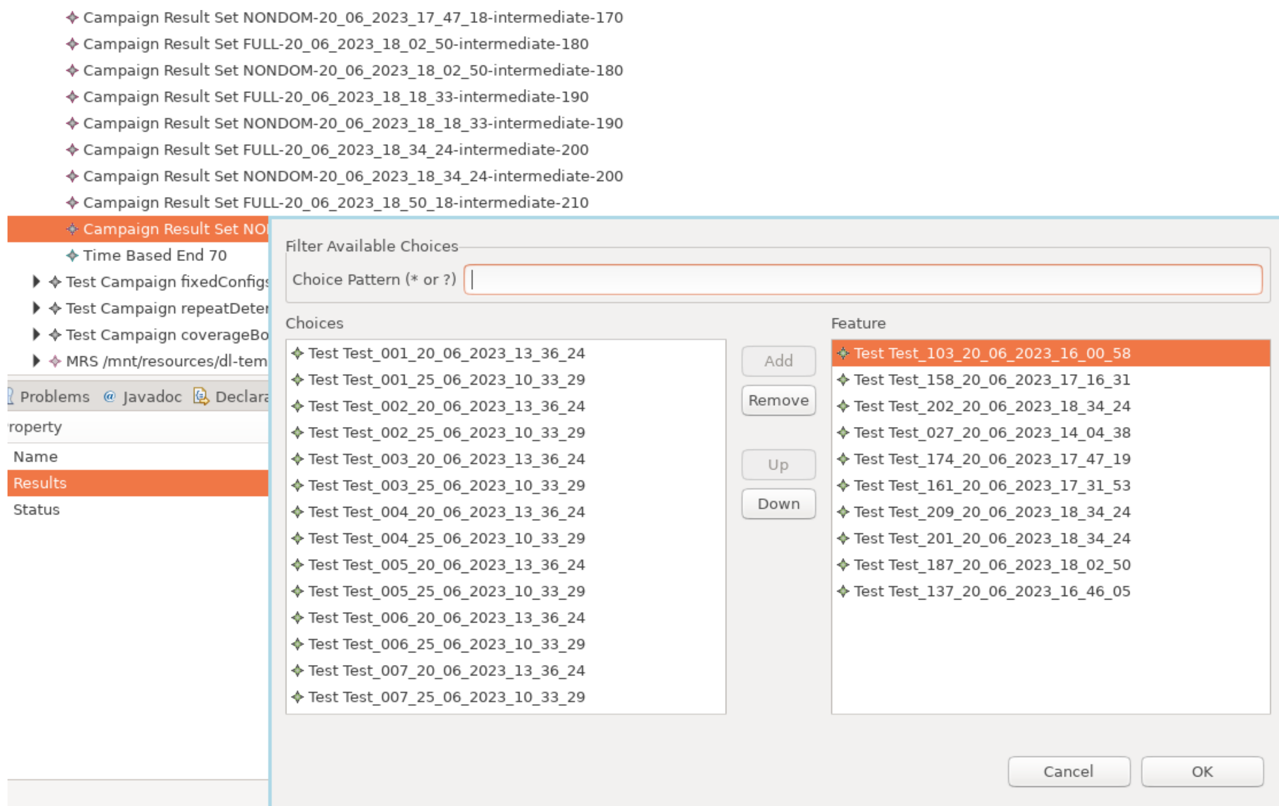


Figure 12: Example result sets for example experiment

A fragment of the output result is shown in Figure 12, showing some result sets from the experiment, with the population of a front at an intermediate generation. The results in the result set are shown here. Using the Exceed editor the tests comprising the result set are shown. Using a provided EGL script under `uk.ac.york.sesame.testing.generator` at **files/resultsAnalysis/resultsAnalysis.egl** allows the results and output metrics to be listed in the Eclipse console window.

Step 4 of Integrated Methodology: In order to perform subset selection for physical testing, a CSV file should be extracted from the model information, to pass to the notebook. A Java executable class is implemented under project `uk.ac.york.sesame.testing.evolutionary`, at *phytestingselection/runner/PhyTestingSubsetSelectionRunner*, which extracts the obtained results to a CSV file (Figure 13). Parameters can be set to choose the model file, and to choose the particular result sets to be extracted to CSV. The CSV file name produced will be based upon the model name, but with values appended giving the chosen result set.

Then, the Jupyter notebook can be loaded by changing to the directory "<GITHUB_ROOT>WP6/notebooks/phytesting/" and executing "jupyter notebook SubsetSelection.ipynb" from a terminal. A fragment of the notebook can be seen in Figure 14. The topmost cell enables providing user-defined parameters (including the CSV input file location), the maximum cost of configurations (which gives the size of subsets selected), and the λ value. The middle cell displayed will execute the subset selection algorithm and visualise the result, and the bottom cell lists the selected configurations for physical execution. The corresponding tests can be looked up in the model and their metrics examined.

Steps 5 and 6 of the Integrated Methodology: The selected tests for physical testing shown in the notebook can be looked up in the model results, and copied from the model to create new configurations with the RepeatedRunner test generation strategy. This will allow them to be tested in physical testing. Physical testing will allow the results to be obtained, potentially repeating the results, and comparing them with simulated execution, to investigate the reality gap. Further details on these steps are given in D6.6 [14] on Steps 5 and 6.

```java
public class PhyTestingSubsetSelectionRunner {
    public static void main(String[] args) {
        ////////////////////////////////////// SET PARAMETERS HERE //////////////////////////////////////////////////////////
        String fileName = "/home/jharbin/runtimeEclipseApplication-sesame/TTSTestProject/models/testingKUKA_TTS.model";
        String campaignName = "coverageExperiment";
        boolean isFinal = false;
        String searchString1 = "NONDOM";
        String searchString2 = "intermediate-210";
        ////////////////////////////////////////////////////////////////////////////////////////////////////////////////

        String csvOut = fileName + campaignName + "reduced-" + searchString1 + "-" + searchString2 + ".csv";

        try {
            List<String> metricNameList = new ArrayList<String>();
            metricNameList.add("collisionOccurance");
            PhyTestingSubsetSelection testPhysub = new PhyTestingSubsetSelectionPartial(new SESAMEStandardDimensionSetReducer(), ne

            testPhysub.loadModelToResults(fileName, campaignName);
            testPhysub.writeOutResultsTabSep();
            testPhysub.writeOutResultsCSV(csvOut);

            System.out.println("Writing out to CSV done...");

        } catch (EolModelLoadingException | InvalidTestCampaign e) {
            e.printStackTrace();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

Figure 13: Extracting the model contents from the result set to process

```python
In [32]: # Set the parameters for the subset selection below
         # MAX_COST - maximum cost value allowed
         # lam - lambda parameter value
         # dims - chosen dimensions for the subset selection

In [39]: csv_filename = "/home/jharbin/eclipse-workspace/TTSTestProject/models/phytesting/results/phytestout-coverageTracking-
         MAX_COST = 4
         lam = 1.0
         dims = ['T1_TIME_MIDPOINT_MEAN', 'P1_PARAMETER_MEAN', 'T2_TIME_LENGTH_MEAN']

In [44]: dim_count = len(dims)
         sim = create_sim_graph_from_data(csv_filename, dims)
         sim_pareto = filter_pareto_only(sim)

         phy, cost = best_subset_via_powerset(sim_pareto, dim_count, lam, MAX_COST)
         print(phy,cost)
         elev = 50
         azim = 120
         ax_dist = 13
         graph_filename = "pw-subsetsel-lambda" + str(lam) + ".pdf"
         plot_graph_with_edges_from_subset(sim_pareto, phy, elev, azim, ax_dist, dims, graph_filename)
```

T1_TIME_MIDPOINT_MEAN 0.8 0.6 0.4
P1_PARAMETER_MEAN 0.0 0.1 0.2 0.3 0.4 0.5
T2_TIME_LENGTH_MEAN 0.0 0.1 0.2 0.3

```python
In [35]: get_tests_by_ids(csv_filename,sim,phy)
Out[35]:
```

	NAME	TESTTAG	T1_TIME_MIDPOINT_MEAN	T2_TIME_LENGTH_MEAN	T3_TIME_MIDPOINT_VAR	P1_PARAMETER_MEAN	P2_PARAM
1	Test_002_20_06_2023_13_36_24	0	0.544386	0.262737	0.099973	0.110493	
3	Test_004_20_06_2023_13_36_24	0	0.616396	0.472328	0.000000	0.216123	
4	Test_005_20_06_2023_13_36_24	0	0.841143	0.244467	0.000000	0.507379	
7	Test_008_20_06_2023_13_36_24	0	0.482577	0.299386	0.096034	0.264252	

Figure 14: SubsetSelection notebook which allows setting parameters, executing and visualising the subset selection, and listing the chosen configurations

3 Hardening and Repairing of Deep Learning Components

This section reports on the developed tool for Hardening and Repairing Deep Learning components as a part of the automated quality assurance of Executable Digital Dependability Identities (EDDI) supported MRS. As such, it describes the realization and usage of the developed technical components and serves as a manual for their use. The information provided in this section complements deliverable D6.4 [18].

3.1 Scope Of Hardening and Repairing Recommendations

Robots while operating in production environments, experience non-stationary changing data distributions. In such scenarios, the trained Deep Learning models (i.e., Deep Neural Networks - DNNs) cannot generalise the learned knowledge (during the training phase) correctly to the operational domain, which results in potential misclassification and prediction errors.

The safety assurance process of Data-Driven and Learning components of EDDIs, introduced in Task 6.1, enables the identification of a set of potential threats to the DNN correctness. Using DEEPKNOWLEDGE's test adequacy criterion proposed in deliverable D6.1 [17], we are able to identify if a new test triggers a novel and potentially erroneous, behaviour of the target DNN model. As a consequence, we are able to assess the DNN's performance in any data domain.

While DEEPKNOWLEDGE and SAFEML [10] can accurately characterize the extent to which a DNN can operate outside of a bounded data domain, new techniques are needed to repair and harden (i.e., improve the quality assurance and testing process) the model when it encounters data shift or concept drift situations (as these situations can lead to making incorrect predictions). To this end, we have identified the following requirements for assuring the safety and correctness of the EDDI-enabled Deep Learning components:

1. Continuously modeling the expected operational domain. To achieve a high standard of testing the tester needs to generate, and execute a large number of tests that conceptually capture and simulate the variations in the real-world environment that causes the model's errors.

2. Continuously adapting the trained model to non-stationary environments.

These requirements have been implemented in a prototype tool for hardening and repairing Deep Learning components. Overall, our tool realises a coverage-guided fuzz testing and repairing technology that can effectively address fundamental problems in DL-based software quality assurance:

1. the Oracle problem and the automated test-case-generation problem. The main difference between GENERATIVEREPAIR and other testing techniques is that the former focuses on the semantic mutation to simulate the operational environment, and the verification of each individual output of the software under test.

2. the correct classification of unseen scenes, which is highly critical for MRS when navigating in a non-stationary environment. GENERATIVEREPAIR is an effective tool for continuous repairing of the DL-based system. An interpretable and explainable incremental learning algorithm has been implemented that allows the detection and learning of unseen scenes without the need for external supervision.

3.1.1 GENERATIVEREPAIR Tool

We implemented the GENERATIVEREPAIR prototype as a self-contained fuzz testing and repairing tool in Python based on the Deep Learning framework Keras (ver.2.1.3) with TensorFlow (ver.2.0) backend. We paid

special attention to developing our tool in a robust multi-paradigm technology using a set of Python components (scripts). Since the EDDI framework is built on top of Python, and the ConSert Monitor is created as a Python WHL binary, then importing Python modules inside the EDDI is a simple process. Python scripts allow flexibility to integrate the tool as a design-time EDDI artifact, which can be deployed into any platform for robotic applications (e.g., ROS). Alongside SafeML, this tool will be integrated into a perception uncertainty monitor fully based on design-time EDDI artifacts.

Our GENERATIVEREPAIR tool is platform-independent, which makes the testing and quality assurance process of any Deep Learning-based system faster and cleaner. Furthermore, GENERATIVEREPAIR is underpinned by a component-based architecture that makes it easily extensible with additional coverage criteria and augmentation techniques.

The underlying conceptual and algorithmic details of this methodology are detailed in deliverable D6.4 [18]. The technical implementation that supports our methodology is described in the following sections.

3.2 GENERATIVEREPAIR Architecture

The general structure of our GENERATIVEREPAIR hardening and repairing technology is illustrated in Figure 15, where two separate but complementary tool-based technologies are collaboratively integrated with other SESAME components to increase the overall MRS performance.

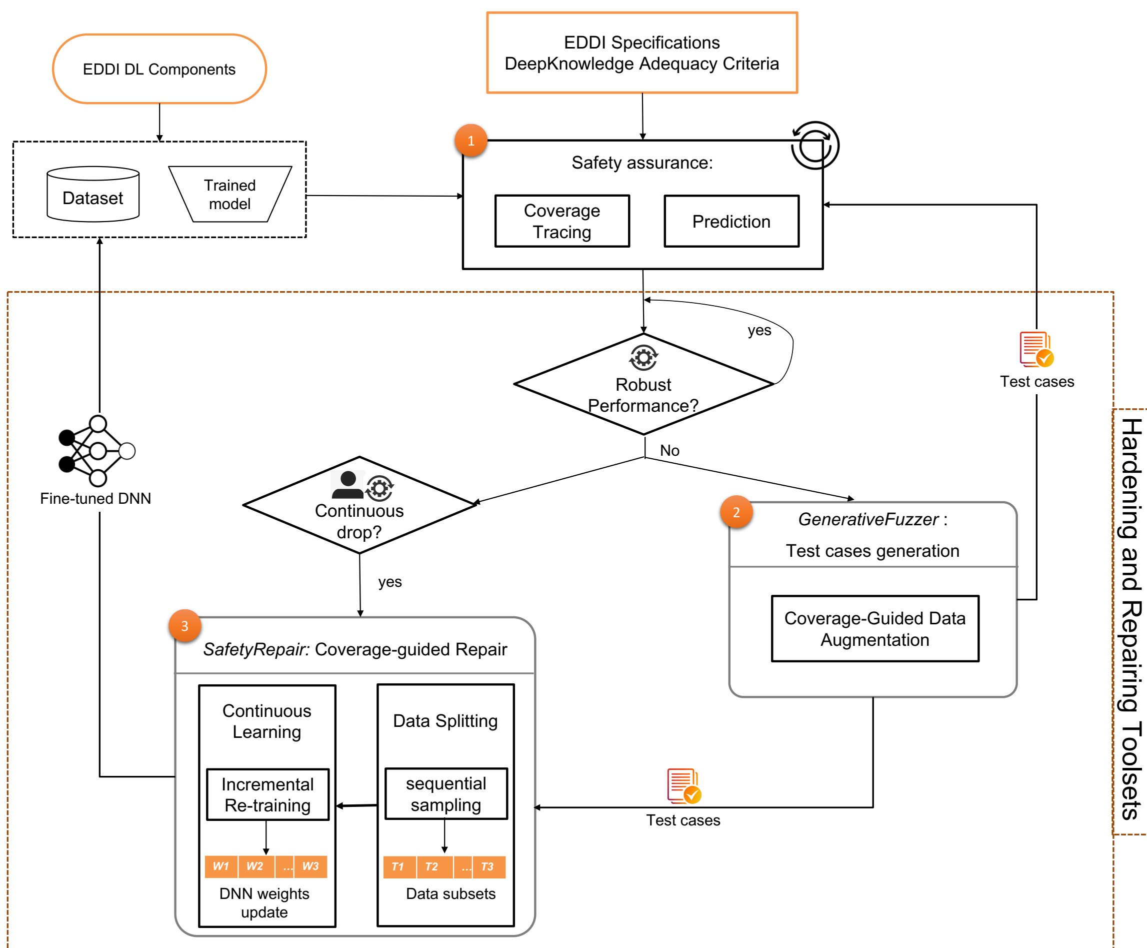


Figure 15: Overview of the Hardening and Repairing Methodology @DesignTime

As illustrated in Figure 15, the hardening and repairing process is based on the safety assurance output results (Step 1). These outputs are automatically analysed and serve as feedback to guide the next steps. GENERA-

TIVEREPAIR iterates between the two main components test case samples generation (GENERATIVEFUZZER), and model repair (SAFETYREPAIR) until it terminates after a fixed number of repair steps. The human icon indicates the need for human assessment or intervention to trigger the next step.

Step1: Safety Assurance. At this step, DL-based software is systematically tested before deployment. The systematic testing of the underlying DNN model helps identify its potential defects and vulnerabilities at an early stage. The safety assurance process provides a concrete set of quantitative metrics for performance assessment.

Performance assessment. At this stage, the quantitative results of safety assurance are statistically analysed. For instance, if the results indicate a high vulnerability to adversarial attacks, and/or insufficient semantic diversity in the test dataset, then the fuzzing operations will be selected and the hyperparameters for the hardening and repairing process will be set up.

Step2: Fuzzing. This step is an automatic test generation process. GENERATIVEFUZZER is a coverage-guided fuzzing component that takes a set of initial seeds and a DNN as the input. It maintains a seed queue and generates passing tests that maximize the coverage criteria, and failing tests that are unsuccessful in activating any of the Transfer-Knowledge Neurons (cf. DEEPKNOWLEDGE approach in D6.1 [17] for more details.). The effectiveness of the fuzzer is affected by the seed selection strategy and the mutation techniques.

Traditional fuzzers mutate a seed with techniques such as flips, block replacement, etc. Our GENERATIVEFUZZER applies semantic mutation (also called data augmentation methods in D6.4), and the mutants, i.e., augmented seeds are filtered based on their photo-realism scores. For each input seed, this process is repeated n times to finally generate a set of k augmented seeds. GENERATIVEFUZZER includes a component that assesses whether an augmented seed triggers an erroneous DNN behaviour, if its prediction result is incorrect. New synthetically-generated test cases are deployed for systematic testing (Step 1).

Performance assessment. After fuzzing, the generated test cases are fed into the DL-based system under test. Our fuzzer is capable of producing synthetic but realistic images and enables simulation of the real-world. Thus, the robustness of the model in the operational environment is assessed by observing its performance, when applied to the synthetic images. If the test cases cause the DNN to misbehave and its performance to drop significantly, users can gather data over a number of iterations and trigger model repair if this is needed.

Step3: Coverage-guided Repair. SAFETYREPAIR focuses on improving the robustness of the DL-based system. Conceptually, this tool improves DNN robustness through fundamentally different continuous learning mechanisms. The selection of the right mechanism depends on the response of the model to the newly generated test cases. Users (software engineers) are provided with a selection of scenarios for continuous training based on the performance monitoring outcomes. This happens sequentially with the test case generation process over a number of iterations to allow an incremental improvement of the DNN model robustness without affecting its performance on the initial dataset. Thus, systematic testing is performed after each repair iteration to ensure the repairing process is achieving the right goal.

3.3 User Guide

This section provides an overview of the implementation of the hardening and repairing tool, its components available for connection to EDDIs and other SESAME components, as well as the structure of the code and an example of execution.

All experiments in our study were conducted on a high-performance computer running a cluster GPU with NVIDIA 510.39. We implemented the GENERATIVEREPAIR methodology and the underlying components based on the state-of-the-art open-source DNN framework Keras (v2.2.2) with Tensorflow (v2.6) as a backend.

GENERATIVEREPAIR is implemented using the Python programming language. All the Python scripts enabling the use and integration of the hardening and repairing tools are available on Github[12] as a set of components linked as shown in Figure 16.

The provided Python scripts can be run into Jupyter Notebook, or in an IDE. In addition, GENERATIVEREPAIR scripts can be run in the command line using the command prompt or PowerShell in Windows. In macOS or Linux, the test engineers can execute GENERATIVEREPAIR using the terminal or xTerm.

The code is available as a set of packages and classes. There are two main scripts "*fuzzing.py*" and "*repairing.py*" respectively dedicated to executing GENERATIVEFUZZER and SAFETYREPAIR.

Coverage Criteria package. This package includes a set of coverage criteria, including DEEPKNOWLEDGE developed in Task 6.1. Our tool currently supports neuron coverage (NC) [15], DeepImportance [6], likelihood-based surprise adequacy (LSA) [11], and the neuron-level criteria k-Multisection Neuron Coverage (KMNC) and Neuron Boundary Coverage (NBC) from DeepGauge [12].

Our tool leverages multiple extensible datasets to fuel the test generation and data augmentation processes. For most of our use cases, the dataset loading operations are available in the $pre - processing.py$ script. The *Dataset* component allows to load and pre-process different datasets including CIFAR-10, COCO, and Grape Leaves for the purpose of fuzzing and repairing. In order to extend the tool and integrate new use cases/datasets, data-loading operations have to be integrated and customized for the targeted dataset following the existing implementations.

This class has different operations for each dataset including:

- *preprocessing()*: which allows to resize and split the dataset according to the user inputs.
- *filtering()*: This operation allows the selection of specific subclasses of the dataset to fit the purpose of training or fine-tuning the subject model.
- *Load_grape_data()*: This operation serves in particular for creating Grape Leaves dataset by combining samples collected from vineyards and digital images from the web, then labeling it using a captioning DL model.
- *augmentsplit()* This operation is used by the repairing component to split the augmented dataset for training in specific time windows for the subject continuous learning paradigm.

Augmentation Package. This package encompasses a set of scripts, which implement different data augmentation methods:

- *tsable_diffusion.py* is an implementation of the text-guided Stable Inpainting method [22]. This script uses other scripts from the *NLP process* package.
- *SemanticOcclusion.py* as its name states is an implementation of the occlusion method [23, 16].
- *GaussianNoise.py* This script implements the metamorphic augmentation technique noise that is used as a baseline in our experimental study.

NLP Process Package. This package includes a set of Natural Language Processing techniques, i.e., *tagging.py*, *captiongeneration.py*, implemented for automatically generating text prompts that serve as input for the Stable Diffusion method.

[12] `https://github.com/sesame-project/MLTesting`

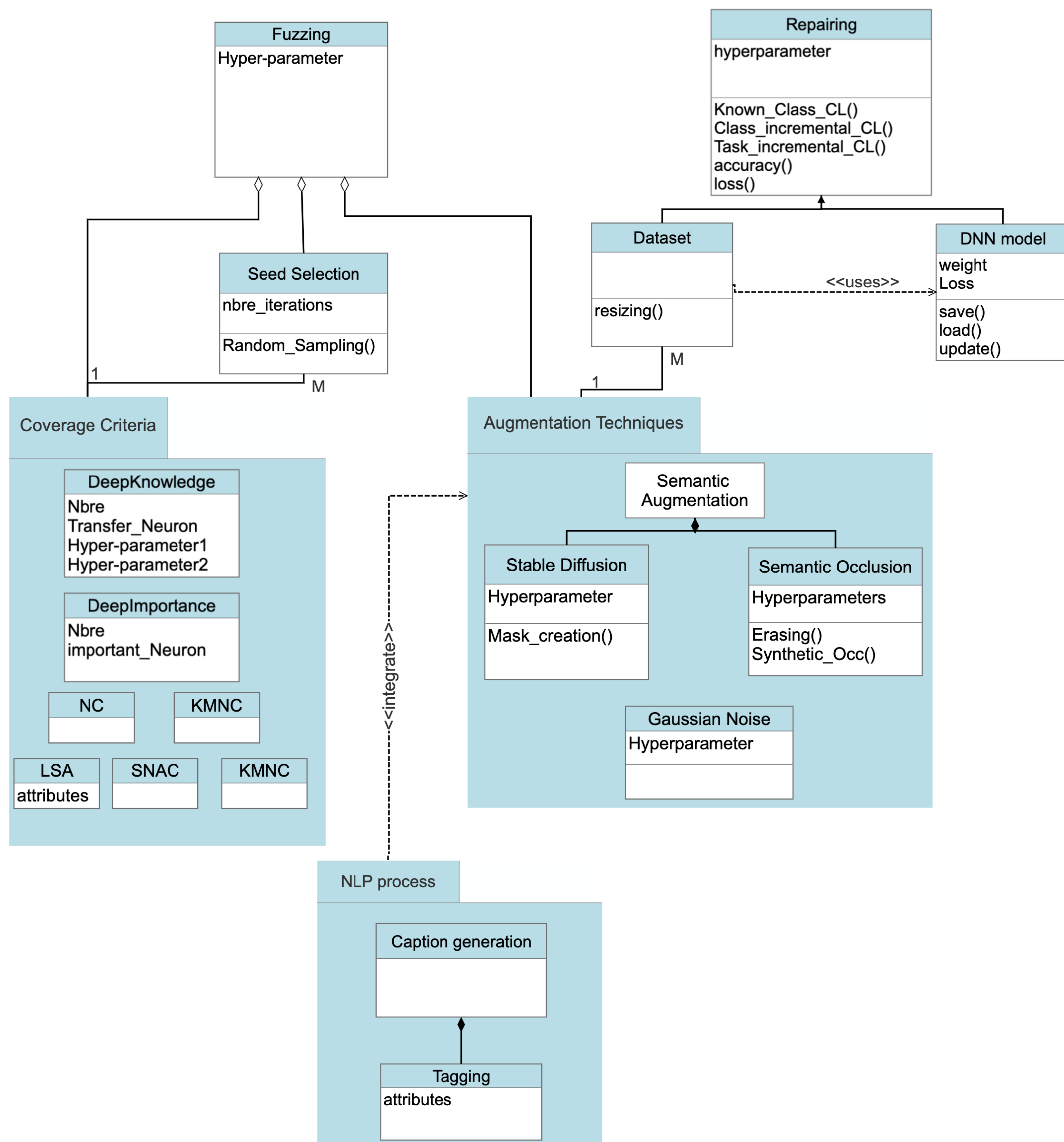


Figure 16: Class diagram for GENERATIVEREPAIR showing dependencies between components.

Repairing Package. This package has a set of scripts for DNN model retraining and repairing. Each script is an implementation of the continuous learning paradigm:

- *CLforKnownClass.py* script is an implementation of the Continuous learning paradigm of known classes.
- *ClassincrementalCL.py* script is an implementation of the Class incremental learning (i.e., new classes) paradigm.
- *TaskincrementalCL.py* this script implements the task incremental learning paradigm, which allows a classical fine-tune operation for the DNN model on new data distribution.

3.3.1 Setting Up the Environment: Required Libraries

The implementation of our GENERATIVEREPAIR approach requires a set of Python, and Tensorflow libraries and different other customized APIs that were used not only to implement our algorithm but also to integrate the state-of-art deployed methods (i.e., coverage criteria, Spacy, etc.). The complete installation guide is available at `https://github.com/sesame-project/MLTesting`. After completing the installation process indicated there, users can continue with the user guide below.

3.3.2 Running the GENERATIVEREPAIR Tool

To run GENERATIVEREPAIR, we can use shell commands, which depend on the hardening and repairing technique chosen:

For coverage-guided fuzzing testing, i.e., GENERATIVEFUZZER, users can execute the commands by running the shell script "*fuzzing.py*" as shown in the following listing.

Listing 1: Example of Shell command to execute GENERATIVEREPAIR

```
#!/bin/bash
Python3.8 fuzzing.py -method[0 or 1] -fuzzer[type_of_fuzzing]  -repair[continuous_learning_paradigm]
    -it[ nbre_of_iteration ] -model [path_to_keras_model_file] -dataset [dataset_name] -approach
    [ coverage_criteria ] - logfile [ path_to_log_file ]
```

For the repairing process, i.e., SAFETYREPAIR, users can execute the commands by running the shell script "*repairing.py*" with the specific parameters detailed in Figure 17.

The user can also run the scripts "*fuzzing.py*", "*repairing.py*" without specifying any of the parameters, given that we set default parameters as shown below:

Listing 2: "*fuzzing.py*" Python script with default parameters

```python

from tensorflow import keras
from tensorflow.keras import applications
# Helper libraries
import argparse
import os
import numpy as np
import matplotlib.pyplot as plt
from datetime import datetime
from Dataprocessing import *
from utils import *
from Coverages.idc import *
from Coverages.neuron_cov import *
from Coverages.tkn import *
from Coverages.kmn import *
from Coverages.ss import *
from Coverages.sa import *
from Coverages.knw import *
from Coverages.TrKnw import *


os.environ['TF_GPU_ALLOCATOR']="cuda_malloc_async"
os.environ["TF_CPP_VMODULE"]="gpu_process_state=10,gpu_cudamallocasync_allocator=10"
 if __name__ == "__main__":

```

```
26            args = parse_arguments()
27            method= args['method'] if args['method'] else 1
28            fuzzer = args['fuzzer'] if args['fuzzer'] else "RInp"
29            iteration = args['it'] if args['it'] else 10
30            repair = args['repair'] if args['repair'] else False
31            model = args['model'] if args['model'] else 'AllConvNet'
32            dataset = args['dataset'] if args['dataset'] else 'Leaves'
33            approach = args['app'] if args['app'] else 'knw'
34            layer = args['layer'] if not args['layer'] == None else -1
35            logfile_name = args['logfile'] if args['logfile'] else 'fuzzing.log'
36            logfile = open(logfile_name, 'a')
37            startTime = time.time()
38
39            Fuzzing(method, fuzzer, iteration, approach, model, dataset, approach, repair, layer)
40            logfile.close()
41            endTime = time.time()
42            elapsedTime = endTime - startTime
```

3.4 GENERATIVEREPAIR Execution Example

This section serves as an illustration of executing the GENERATIVEREPAIR tool. The example shows the application of our tool to the viticulture use case. The spraying mission could be achieved using LXSNS's DNN algorithms with the novel MRS capabilities for sensor fusion. To this end, high importance is set on DNN accuracy and safe behaviour.

The successful accomplishment of fungicide spraying depends on the optimal functioning of DL-based software that is able to detect the affected plants and correctly classify the detected disease. The correct execution of these actions will enable the provision of the right treatment. This is considered a key functional requirement for this use case. Besides its performance, the adapted DNN model needs to detect a wide range of diseases that are not fully represented in the collected dataset from the vineyards.

In this example, we showcase how to run the "*repairing.py*" script for hardening and repairing the Grape Leave disease detection model (cf. Figure 19). This task is performed with the AllConvNet model trained on images whose size is 32x32.

We start by creating and starting the virtual environment. In our case, we activate the virtual environment $SESAME$ under the project folder EU_SESAME.

Listing 3: Creating and starting the virtual environment

```
1
2 $ mkdir EU_SESAME
3 $ cd EU_SESAME
4 $ virtualenv SESAME
5 $ source SESAME/bin/activate
```

- **-method :** an integer took 0 for fuzzing operation and 1 for repairing.
- **-fuzzer** : the name of the selected fuzzer. Our platform provides 5 strategies, with the possibility of chose a combination of different testing criteria as guidance. These fuzzers are:
  - Random Noise testing (**RN**). This combines random sampling as seed selection strategy and Gaussian Noise for data augmentation
  - Random Inpainting (**RInp**). This fuzzer uses random sampling strategy and test-guided Stable Diffusion Inpainting as data augmentation.
  - Random Semantic Occlusion (**SemOcc**). This fuzzer uses random sampling strategy with Semantic occlusion (both random erasing and synthetic occlusion similarly) to augment each input seed.
  - DeepKnowledge Inpainting (**KnwInp**). Different from **RInp**, this strategy guides testing using deepKnowledge coverage criteria as feedback. An input seed is put to the seed queue if it improves the deepknowledge coverage.
- **-repair** : this the selected paradigm for continuous learning. We can select :
  - CLTask : Task incremental learning
  - CLClass : Class incremental learning
  - CL : Continuous learning of known classes
- **-it :** number of iteration for data augmentation within the fuzzing process. We advice to select an integer between 2 an 10.
- **-app:** for approach. **T**he approach for coverage estimation. The selected coverage is used as guidance in each iteration to pick the augmented seed as new test. Our current implementation supports DeepKnowledge (Knw), DeepImportance (idc),(nc),(kmnc),(nbc),(snac),(tknc),(ssc), (lsa), and (dsa).
- **-model :** the name of the Keras model file. The trained keras model is saved as .hdf5 file or the architecture can be saved as JSON and the weights
  saved separately as an .h5 file. All the trained model are saved under the folder `Networks'. Our implementation provides three trained DNN models including **Allconvnet.h5**, **LeNet5.h5**, and **Vgg19.h5**.
- **-dataset :** name of the dataset to be used. Current implementation supports Cifar-10 (**cifar**) COCO (**coco**) and grape leaves (**grape**). Our platform is extensible and other dataset can be added by modifying `**Dataprocessing.py**' and `**Data_Augment.py**' scripts.
- **-layer** : The subject layer's index for approaches including 'idc','tknc', 'lsa'. Note that only trainable layers can be selected.
- **- logfile** : The name of the file that the results to be saved.

Figure 17: Parameters for configuring GENERATIVEREPAIR

Once, all the necessary packages are loaded through the virtual environment, we apply the methodology presented in Section 3.2 (Step3) by running the shell script with the default settings (Listing 4).

The execution of "*repairing.py*" at first involves running the '*Random + Inpainting*' fuzzer to enable the generation of the augmented set for the model retraining. As shown in Figure 20, a set of 32x32 coloured synthetic images are generated and evaluated for photo-realism before being used for the retraining paradigm.

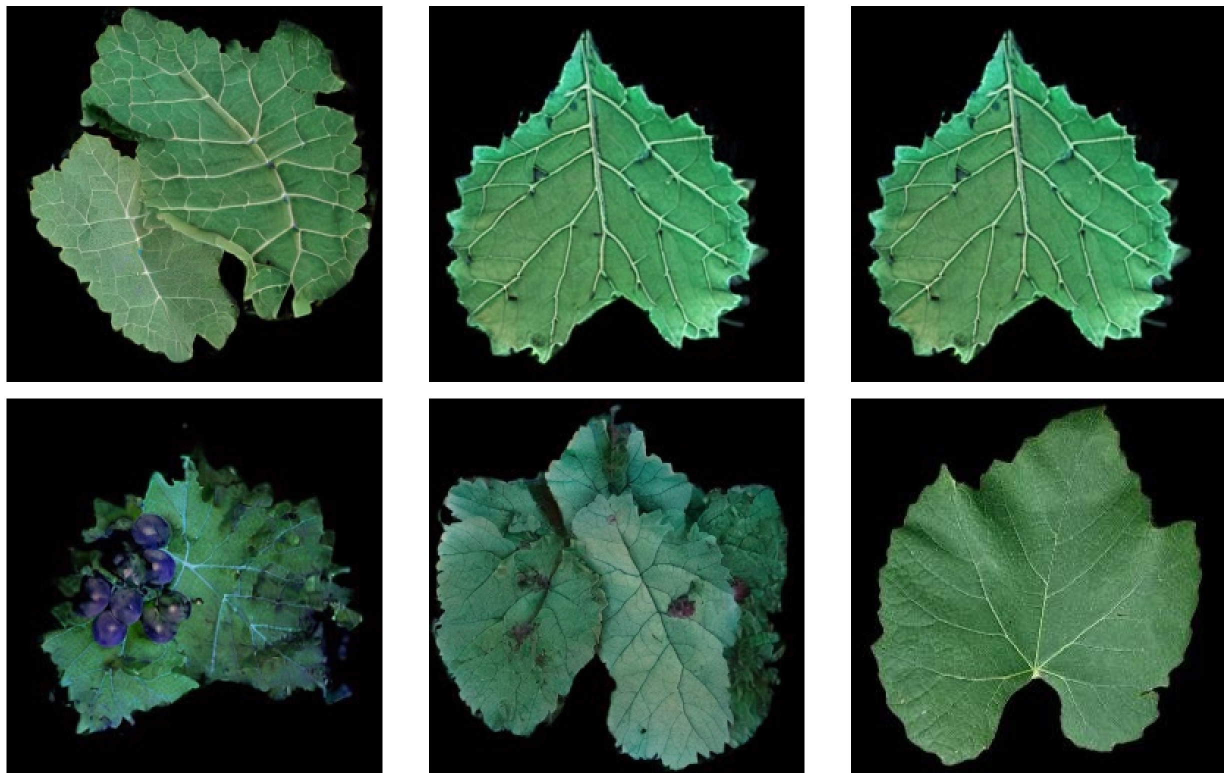


Figure 18: Examples of the repairing outputs. A set of synthetic images generated using Random+Inpainting fuzzer. All these images are augmented using the same input seed.

Listing 4: Set of default parameters run the Python script "fuzzing.py"

```
1
2 −method = 0 (  fuzzing  )
3 −fuzzer  = RInp Random Inpainting
4 −repair  = CLTask
5 −it  = 10
6 −app= Knw
7 −model = AllConvNet
8 −dataset  =grape
9 −layer  = −1
10 −  logfile  = fuzzing . txt
```

Figure 18 showcases a small set of augmented images generated using '*Random + Inpainting*' fuzzer with the Grape Leaves dataset as an input set. After fidelity estimation, these augmented images serve as new test cases for model repair.

The model is then fine-tuned using $CLTask$, i.e., using the task incremental learning paradigm. To this end, we have trained the initial version of AllConvNet model to detect the Esca vascular wilt disease that attacks the perennial organs of grapevines. With the help of the augmented set, we fine-tune the model to detect different types of disease. For the first run (cf. Figure 20), we generate 1915 synthetic images that have been filtered and the remaining set of 1232 images is used to train the model on two different types of diseases, including Isariopsis Leaf Spot and Black Measles.

The fine-tuned model is then saved in the Network folder with a combination of the original model name and the fuzzer name as the name of the new extension of the model at time t.

```
Terminal Shell Edit View Window Help
sondessmissaoui —
The default interactive shell is now zsh.
.
.york.ac.uk
Welcome to Ubuntu 18.04.6 LTS (GNU/Linux 5.4.0-89-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage
Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

Last login: Mon Jun 19 11:54:18 2023 from 10.240.223.219
~$ conda deactivate
~$ cd sm2672
2$ source SESAME/bin/activate
(SESAME) ~/sm2672$ python3.8 R
Regym/        REGYM/        Regymbackup/
(SESAME) ~/sm2672$ cd EU_SESAME
  [Bookmarked 26 Jun 2023 at 10:05:15]
  [Bookmarked 26 Jun 2023 at 10:05:18]
(SESAME) ~/sm2672/EU_SESAME$ python3.8 Repairing.py
```

Figure 19: Shell command to execute "*repairing.py*" script- 1

```
accuracy on only augmented  noise test set: 12.20%
2/2 - 0s - loss: 6.3236 - accuracy: 0.3774 - 18ms/epoch - 9ms/step
NOISE+ orig, accuracy: 37.74%
 python3.8 Repairing.py
2023-06-26 10:10:26.858939: W tensorflow/core/platform/cloud/google_auth_provider.cc:184] All attempts to get a Google
authentication bearer token failed, returning an empty token. Retrieving token from files failed with "NOT_FOUND: Could
 not locate the credentials file.". Retrieving token from GCE failed with "FAILED_PRECONDITION: Error executing an HTTP
 request: libcurl code 6 meaning 'Couldn't resolve host name', error details: Could not resolve host: metadata".
Found 400 images belonging to 10 classes.
Found 50 images belonging to 10 classes.
Found 50 images belonging to 10 classes.
100%|████████████████████████████████████████| 12/12 [00:00<00:00, 32.14it/s]
100%|█████████████████████████████████████████| 1/1 [00:00<00:00, 56.73it/s]
100%|█████████████████████████████████████████| 1/1 [00:00<00:00, 49.69it/s]
shape  (200, 32, 32, 3) (200, 10) (50, 32, 32, 3) (50, 10) (200, 32, 32, 3) (200, 10)
test set size: 50
 loaded from  /home/sondess/sm2672/EU_SESAME/experiments/coco_topgained_testset_knw.sav
size of test knw preffered 2128
load augmented data ...
change reshape
Found 1915 images belonging to 10 classes.
Found 25 images belonging to 10 classes.
Found 2050 images belonging to 10 classes.
100%|████████████████████████████████████████| 59/59 [00:01<00:00, 34.27it/s]
100%|█████████████████████████████████████████| 1/1 [00:00<00:00, 50.76it/s]
100%|████████████████████████████████████████| 64/64 [00:01<00:00, 33.00it/s]

 size of synthetic set for retraining
  (1915, 32, 32, 3) (1915, 10)
exeception
2023-06-26 10:10:32.567505: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:975] successful NUMA node read from
SysFS had negative value (-1), but there must be at least one NUMA node, so returning NUMA node zero
2023-06-26 10:10:32.588277: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:975] successful NUMA node read from
SysFS had negative value (-1), but there must be at least one NUMA node, so returning NUMA node zero
2023-06-26 10:10:32.588408: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:975] successful NUMA node read from
SysFS had negative value (-1), but there must be at least one NUMA node, so returning NUMA node zero
2023-06-26 10:10:32.588679: I tensorflow/core/platform/cpu_feature_guard.cc:193] This TensorFlow binary is optimized wi
```

Augmented dataset

Figure 20: Shell command to execute "*repairing.py*" script- 2

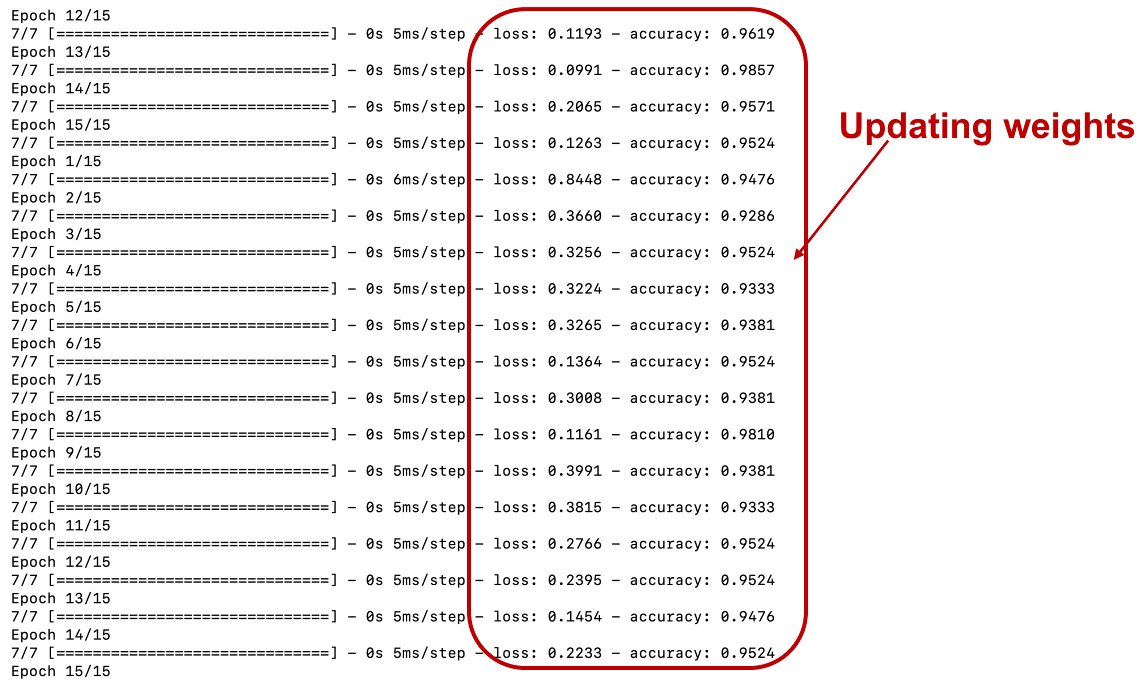


Figure 21: Shell command to execute "*repairing.py*" script- 3

As a final note, the previous results showcase the importance of continuous hardening and repairing of Deep Learning-based software components in MRS. The results illustrated in Figure 21, demonstrate the effectiveness and efficiency of our tool in repairing the LXSNS's DNN algorithm. It helped achieve an accuracy of 95.24% for the AllConvNet model after retraining on the set of synthetic images generated by our tool GENERATIVEFUZZER, in particular, the 'Random+Inpainting' fuzzer.

Moreover, our tool leverages the outstanding capabilities of semantic data augmentation for Deep Learning repairing and continual learning in an easily extensible way that currently can be adapted as a design-time artifact.

4 Conclusion

This deliverable has presented the final version of the quality assurance toolset, including quality assurance tools for data-driven learning components, and simulation-based testing tools for EDDIs. This toolset has been developed in the context of Tasks 6.3-6.5, and follows deliverable D6.3 which reported the developments of Tasks 6.1 and 6.2. The underlying architecture and scientific innovations have been presented in the other WP6 deliverables. As such, this document complements deliverables D6.4 and D6.6.

The developed prototype tools are compatible with most systems and architectures used by our industrial partners. The development process has followed the agreed architectural conventions specified for the integrated platform of SESAME and the partners' requirements from deliverable D1.1.

The simulation-based testing toolset implements the methodologies and architectures (presented in deliverable D6.6 [14]) for integrated testing, beginning with simulation-based testing and informing the transition between simulation-based testing and lab experimentation. We presented the interfaces and DSL structure for these simulation-based testing components, together with a usage example for these technologies.

The quality assurance tool for Data-Driven and Learning components implements the methodologies and architectures presented in deliverable D6.4 [18] for (1) the GENERATIVEFUZZER tool, i.e., a coverage-guided fuzzing technique; and (2) SAFETYREPAIR tool, i.e., the repairing tool that serves as a continual learning technique for DNNs. Both tools are implemented and integrated into a self-contained fuzz testing and repairing framework named GENERATIVEREPAIR, implemented as a set of Python components. These components allow test engineers and system integrators to customise the testing and repairing operations and pass in parameters from the command line. We provided a comprehensive presentation of our tools and gave step-by-step usage examples.

References

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

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] Simos Gerasimou, Hasan Ferit Eniser, Alper Sen, and Alper Cakan. Importance-driven deep learning system testing. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 702–713. IEEE, 2020.

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

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

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

[10] Fraunhofer IESE and University of Hull. D7.2 tools for generation of runtime eddis., 2022.

[11] Jinhan Kim, Robert Feldt, and Shin Yoo. Guiding deep learning system testing using surprise adequacy. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1039–1049. IEEE, 2019.

[12] Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, et al. Deepgauge: Multi-granularity testing criteria for deep learning systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 120–131, 2018.

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

[14] SESAME Project Partners. D6.6: Multi-stage quality assurance methodology for EDDIs. Technical report, The Open Group, Jun 2023.

[15] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *proceedings of the 26th Symposium on Operating Systems Principles*, pages 1–18, 2017.

[16] István Sárándi, Timm Linder, Kai O Arras, and Bastian Leibe. Synthetic occlusion augmentation with volumetric heatmaps for the 2018 eccv posetrack challenge on 3d human pose estimation. *arXiv preprint arXiv:1809.04987*, 2018.

[17] Missaoui Sondess, Gerasimou Simos, and Matragkas Nicholas. D6.1 assurance of data-driven and learning components of eddis., 2022.

[18] Missaoui Sondess, Gerasimou Simos, and Matragkas Nicholas. D6.4 recommendations for EDDIs hardening and repairing, June, 2023.

[19] 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.

[20] 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.

[21] 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.

[22] Anwaar Ulhaq, Naveed Akhtar, and Ganna Pogrebna. Efficient diffusion models for vision: A survey. *arXiv preprint arXiv:2210.09292*, 2022.

[23] Zhun Zhong, Liang Zheng, Guoliang Kang, Shaozi Li, and Yi Yang. Random erasing data augmentation. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 13001–13008, 2020.