Intelligent Verification/Validation for XR Based Systems

Research and Innovation Action

Grant agreement no.: 856716

D3.3 – 2nd prototype of Functional Test Agents (FTAs)

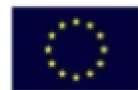
iv4XR – WP3 – D3.3

Version 1.8

December 2021

Project Reference	EU H2020-ICT-2018-3 - 856716
Due Date	31/12/2021
Actual Date	XX/12/2021
Document Author/s	Tanja Vos (UPV), Wishnu Prasetya (UU), Fitsum Kifetew (FBK), Fernando Pastor Ricós (UPV), Alexandre Kazmierowski (THA-SIX), Joseph Davidson (GA), Pedro Fernandes (INESC-ID), Jeremy Cooke (GWE)
Version	1.8
Dissemination level	Public
Status	Draft
Type	OTHER

This project has received funding from the European Union's Horizon 2020 Research and innovation programme under grant agreement No 856716



Document Version Control			
Version	**Date**	**Change Made (and if appropriate reason for change)**	**Initials of Commentator(s) or Author(s)**
1.0	17/11/2021	Initial document structure and contents	TV, FP
1.1	2/12/2021	Discuss content and add tasks description	TV, WP, FK, FP, AK, JD, PF, JC
1.2	8/12/2021	Added Task 3.4 Coverage section	FK
1.3	9/12/2021	Added Task 3.1 Specifying Tests sections	WP
1.4	9/12/2021	Added Task 3.5 Multi Agent section	FP, WP, AK, JD, PF
1.5	13/12/2021	Added Task 3.2 Explorative FTA section	FP
1.6	14/12/2021	Added Task 3.2 Goal Solving FTA section	WP
1.7	15/12/2021	Update Task 3.2 Goal Solving and Task 3.4	AK
1.8	16/12/2021	Update task 3.3	TV
1.9	17/12/2021	Overall Integration picture	TV, FP

Document Quality Control			
Version QA	**Date**	**Comments (and if appropriate reason for change)**	**Initials of QA Person**
1.0	XX/12/2020	Initial comments and section assignment	
1.9	29/12/2021	Few corrections and comments	AS

1.9	29/12/2021	Review and edits	IS

Document Authors and Quality Assurance Checks

Author Initials	Name of Author	Institution
TV	Tanja Vos	UPV
WP	Wishnu Prasetya	UU
FK	Fitsum Kifetew	FBK
FP	Fernando Pastor	UPV
AK	Alexandre Kazmierowski	THA-SIX
JD	Joseph Davidson	GA
PF	Pedro Fernandes	INESC-ID
JC	Jeremy Cooke	GWE
AS	Angelo Susi	FBK
IS	Ian Saunter	GWE

TABLE OF CONTENTS

Executive Summary 5
Acronyms and Abbreviations 5
Overall concepts, architecture, design Functional Test Agents (FTAs) 6
Task 3.1 : Specifying Tests 7
Task 3.2 : Goal Solving Agents 10
Task 3.2 : Exploration Agents 13
Task 3.3 : Dealing with Hazardous Elements 18
Task 3.4 : Coverage 19
Task 3.5 : Multi Agent Testing 22

EXECUTIVE SUMMARY

This deliverable D3.3 is of type OTHER, it describes the status of the 2nd prototype of Functional Test Agents (FTAs) and the progress made during the second year related to the FTAs. The real delivery is the software which is available on a github repository.

In this deliverable we will summarize the overall concepts, architecture, design, and technical choices. With the intent to give a clear overview for the reviewers of the work that has been done in WP3, we will describe per task:

- Short introduction to the task
- What has been done
- Where the result can be found (link to github, zenodo, videos) and how to use them
- What is planned for the 3rd year

ACRONYMS AND ABBREVIATIONS

FTA	Functional Test Agent
SUT	System Under Test
XR	Extended Reality
AI	Artificial Intelligence
WOM	World Object Model
API	Application Programming Interface
RL	Reinforcement Learning
DRL	Deep Reinforcement Learning
DSL	Domain Specific Language
TSL	Test Specification Language
LTL	Linear Temporal Logic
MDP	Markov Decision Process
MBT	Model Based Testing

OVERALL CONCEPTS, ARCHITECTURE, DESIGN FUNCTIONAL TEST AGENTS (FTAS)

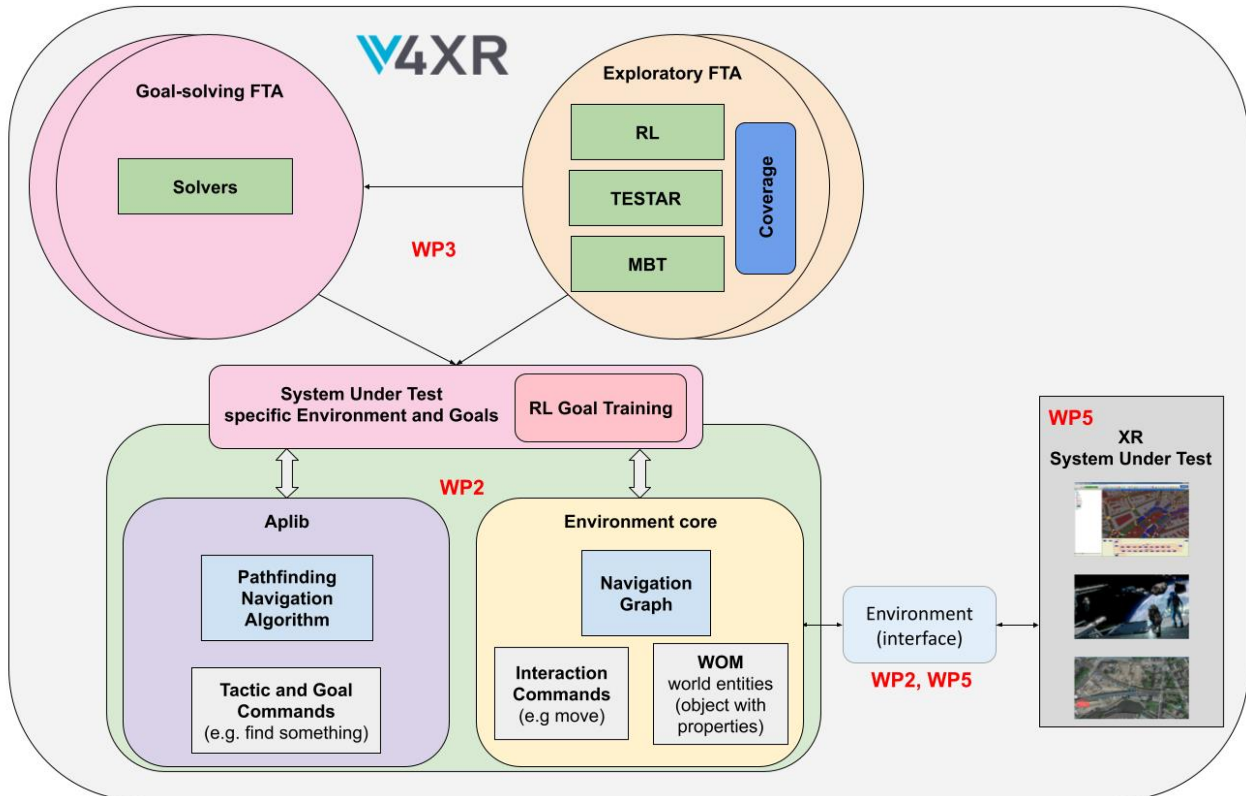


Figure 1: Different types of FTAs in iv4XR.

Two types of FTA's are being distinguished in WP3:

- The first type of agent makes deliberations to choose the appropriate strategies that will allow it to do ***goal-solving*** (the left circle in Figure 1).
- The second type of agent is intended to test the functionality of the XR system using ***exploration*** (the right circle in Figure 1).

These FTAs are able to test the SUTs from WP5, by using the ApLib and the Core from WP2, as is also shown in Figure 1.

TASK 3.1 : SPECIFYING TESTS

<table>
<tr><td>Short introduction to the task</td><td>This task aims to develop a test specification language that would allow developers to abstractly specify complex testing tasks.</td></tr>
<tr><td>What has been done until the second year: results</td><td>The test specification language has been implemented as a domain specific language (DSL) embedded in the language Java. It has also been tested with various case studies.

An example of a testing task formulated using this DSL is shown below[1]:

<pre>SEQ(interacted(e1),
    interacted(e2),
    refreshed(d),
    assertTrue_(world → world.getEntity(d)
                              .getProperty("isOpen"))
)</pre>
The task says that the agent should interact with an in-world entity e1 and then e2, and then it should refresh its knowledge of the state of another entity called d (e.g. by walking to a place near to d so that it can see it to refresh what it knows about it). And then the agent should check the asserted property, namely that this d should be “open” (e.g. d could be a door, which can be opened and closed). If the assertion holds then the test is passed, else it is violated.

We can see that the formulation of a testing task consists of two parts:
1. The part for specifying what to test (blue); that is, for specifying the correctness properties that should hold..
2. The part for specifying how to test (yellow); that is, for specifying high level steps to be taken to take the SUT to states where it is sensical to sample the correctness properties.

To achieve abstraction, from the testers’ perspective tasks are formulated declaratively as goals. A goal only specifies what needs to be achieved. E.g. in the above example, interacted(e) specifies that the entity e should be interacted; it does not say how to get to e in the first place, which may involve some complex navigation through a maze. But indeed, the executing test-</td></tr>
</table>

[1] The example is presented in a more abstract syntax than the actual concrete syntax. Samples of actual tests can be found in the iv4xrDemo project (https://github.com/iv4xr-project/iv4xrDemo), e.g. in the test src/test/java/agents/demo/RoomReachabilityTest.java.

	agent will ultimately have to know how to solve the goals given to it. This is discussed in Task 3.2. Further abstraction can be obtained by having an algorithm that can fill in a large part of the how-to-test part (the yellow part in the above example). This is also discussed in Task 3.2, e.g. by using an on-line search algorithm, As a new feature, we have also added Linear Temporal Logic (LTL)[2] to the test specification language. In the previous testing task example (above), the correctness property is formulated as a *state predicate*, stating that the state of the in-world entity d should be as such that its *isOpen* property is true when the agent observes it. In contrast, an LTL formula is a predicate over a sequence of states, e.g. this could be the sequence of states that are passed during the execution of a testing task. With such a property we can assert that throughout the execution the value of a certain variable is always positive, or that the value of another variable should start as 0 but eventually it should become positive. Proving LTL is essential for WP4, because often emotion is not a property of a state, but rather a property of an execution (e.g. the agent should eventually feel a certain emotion[3]).
Where are these results and how to use them	**User Documents.** The Test Specification Language is actually part of the DSL for formulating goals and tactics for agents, so they share the same underlying concepts. More on the concepts of iv4xr agent programming can be found in its Documentation page: https://github.com/iv4xr-project/aplib/blob/master/README.md **User Reference.** 1. The syntax of the DSL test specification language: https://github.com/iv4xr-project/aplib/blob/master/docs/manual/DSL.md 2. APIs Reference is provided as part of the APIs reference of the Framework-core: http://www.staff.science.uu.nl/~prase101/research/projects/iv4xr/aplib/apidocs/ The key classes implementing the DSL are AplibEDSL and iv4xrEDSL in the package nl.uu.cs.aplib and eu.iv4xr.framework. The implementation of LTL can be found in the class LTL and its subclasses in the package eu.iv4xr.framework.extensions.ltl. We also provide an extension of LTL called Bounded-LTL, which can be found in the class BoundedLTL in the same package

[2] For more about LTL, see e.g. Baier, Christel, and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008.

[3] If *f* is a state predicate, In LTL "Eventually" *f* is a sequence predicate that says that some state in the sequence satisfies f, without being specific which state should that be (it does not have to be the current state, or the next state etc, as long as some state in the sequence satisfies *f*).

<table>
<tr><td></td><td>

Usage Examples

1. A starting (simple) example of using the DSL is provided in the above Documentation page of the Framework. Direct link: https://github.com/iv4xr-project/aplib/blob/master/docs/iv4xr/testagent_tutorial_1.md

2. Examples of the use of LTL are provided in the above Documentation page of the Framework. Direct link: https://github.com/iv4xr-project/aplib/blob/master/docs/iv4xr/testagent_tutorial_3.md

3. A full example in the context of a realistic SUT can be found as part of the iv4xrDemo project that demonstrates the use of iv4xr to test a 3D maze-puzzle game. The Demo project can be found here: (https://github.com/iv4xr-project/iv4xrDemo. Look for example in the test-example src/test/java/agents/demo/RoomReachabilityTest.java.

Implementation.
The Test Specification Language is part of the Framework-core, which can be obtained here: https://github.com/iv4xr-project/aplib

For inspecting the code, the key classes that implement the DSL are AplibEDSL and iv4xrEDSL. The implementation of LTL and Bounded-LTL can be found in the classes LTL and BoundedLTL.

Publications:
Aplib: Tactical Agents for Testing Computer Games. Paper by Prasetya, Dastani, Prada et al. Published in EMAS 2021, https://link.springer.com/chapter/10.1007/978-3-030-66534-0_2

</td></tr>
<tr><td>

Task Status overview and plan for the next year

</td><td>

The table below shows the task status and plan for next year.

☐ Completed. Only minor changes are expected for next year.
☐ Implemented, but need possibly major improvement.

<table>
<tr><td>D3.3 status</td><td>Result (type)</td><td>Planned for D3.4</td></tr>
<tr><td>Completed</td><td>Embedded language for formulating test goals and tactics. (software)</td><td></td></tr>
<tr><td>Completed</td><td>Embedded language for formulating assertion-like correctness properties. (software)</td><td></td></tr>
<tr><td>To be improved</td><td>Embedded language for specifying temporal correctness properties. (software)</td><td>Investigating the use of temporal properties to formulate emotion-</td></tr>
</table>

</td></tr>
</table>

			related requirements (together with WP4)

TASK 3.2 : GOAL SOLVING AGENTS

Introduction to the task	This task aims to develop algorithms to solve goals that are given to a test agent. Test-related goals are usually related to getting the SUT to a certain state that must be tested, or getting the test agent to a certain location in a virtual world. One should take into account that the virtual world can be pretty complex, with non-trivial layout and elements that can dynamically block access to certain states. Without automated goal solving, the developers will have to manually provide the sequence of subgoals to guide the agent to its final goal. This can be tedious if the sequence is long, and has to be manually repaired as well if the SUT is changed. With respect to testing tasks discussed in T3.1, we can think of two levels where solvers are used: step-level and task-level. Recall that a testing task consists of a sequence of high-level steps followed by an assertion of the correctness condition to check. Each high level step is essentially a goal for the test-agent, which may be non-trivial to solve, and therefore having a solver would be useful. Pathfinding related solvers are examples of typical step-level solvers. In addition to this, we also consider solvers that can construct the sequence of steps itself so that the tester only needs to specify what to test, and leaving it to the agent to figure out the sequence of steps needed to get to a relevant state on its own. For example, an online solver can do this (see below), whereas a learning based solver can be used for solving goals in both levels, as long as there is enough data to train the solver. The following types of solvers are being developed in T3.2: ● **Graph-based pathfinding and exploration algorithm**. A *pathfinding* algorithm is used to plan a path in a virtual world. A test agent can then follow the path to auto-navigate from one location in the world to another. A graph-based algorithm requires a graph to be constructed, representing the navigable parts of the world, but once such a graph is available, pathfinding can be done very efficiently. An *exploration* algorithm is used to explore the graph. This is useful when the agent

<table>
<tr><td></td><td>needs to find a certain entity in the world, if its location is not known upfront[4].

Reasoning-based solver, e.g. using Prolog as a reasoning engine. The use of Prolog is pretty common in implementation of intelligent agent frameworks, e.g. as in GOAL and 2APL. Prolog allows reasoning rules to be formulated, thus allowing a running agent to use them for making decisions. Among other things, this can be a decision to deploy a new subgoal that would help the agent to solve its main goal.
Model-based solvers. This kind of solver can solve a goal provided a behavior model M is given. This is still under development and is scheduled to be finished in year-3.
Online solvers. An online solver tries to solve a goal by actually trying different interactions on the SUT, usually following a certain heuristic. While an online solver is computationally more expensive, it has an important benefit that it does not need a model. We will also investigate its use for constructing a behavior model, and hence allowing model based solvers to be used for subsequent testing tasks (thus combining the advantages of both approaches).
Learning algorithm. A SUT-specific problem can be expressed as an environment for Reinforcement Learning by formalizing it as a Markov Decision Process (MDP) with states, actions and reward. Developers need to implement this formalization with the commonly used (OpenAI) Gym interface. Then, a Deep Reinforcement Learning (DRL) agent can be used to solve the SUT-specific goal. Whereas DRL agents are generic and adapt to many definitions of states, actions and rewards, they are computationally extensive during their training phase and may require careful parameter tuning. On the other hand, the DRL agent will explore the SUT freely during its training phase, in a way that was not planned by the developer. This can prove useful for coverage analysis, or to learn complex behaviors.
</td></tr>
<tr><td>What has been done until the second year: results</td><td>
A* Pathfinding has been implemented in the Framework, and along with it a frontier-based graph exploration algorithm. While these algorithms are not not novel, their use in the context of automated testing is actually not much discussed in the literature, so we also wrote a paper about this subject; see below.
A Prolog engine has been integrated to iv4xr agents so that it can be used for goal solving over a model M, if the model can be encoded as Prolog facts and rules.
</td></tr>
</table>

[4] Even if the location is known, we may not want to rely on its location to find it as this location may change when developers decide to change the world (which happen often, at the development time). Tests are more robust if they are not location-fixed.

	● A Model Based Testing (MBT) package has been developed. The package allows models to be formulated in terms of extended Finite State Machines (EFSM). The package includes implementation of a number of search algorithms, serving as model-based solvers. A study on its performance (as a solver) has been conducted. The results are published (see below). ● An initial online solver algorithm was designed and implemented. Its performance has been studied and the results are published (see below). ● A plugin to the iv4xr core Framework aimed at defining Deep Reinforcement Learning environments with the SUT has been developed. It is compliant with the widely used Gym interface in RL Research. An RL environment for the THA-AVS powerplant intrusion pilot has been developed thanks to this plugin. The TD3 algorithm has been implemented and adapted to solve a simplified version of this environment that shares the same state, observation and reward model. The results are published (see below).
Where are these results and how to use them	● **Graph-based pathfinding (A*) and exploration** are implemented in the Framework-core. The Framework-core can be obtained here: https://github.com/iv4xr-project/aplib **APIs Reference** is provided as part of the APIs reference of the Framework-core: http://www.staff.science.uu.nl/~prase101/research/projects/iv4xr/aplib/apidocs/ The key classes are Navigatable and AStar in the package eu.iv4xr.framework.extensions.pathfinding. Automated exploration is provided by the class SurfaceNavGraph in the same package. **Paper:** *Navigation and Exploration in 3D-Game Automated Play Testing*, by Prasetya, Volhol, Tanis, et al., in the proceedings of the 11th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation, co-located at the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), 2020. Pdf: https://arxiv.org/pdf/2009.07015 ● **Prolog engine.** A prolog engine has been integrated into the iv4xr Framework. The Framework can be obtained here: https://github.com/iv4xr-project/aplib

<table>
<tr><td></td><td>
APIs Reference is provided as part of the APIs reference of the Framework-core. This temporarily hosted here (waiting for a better solution, e.g. to be hosted in Jitpack): http://www.staff.science.uu.nl/~prase101/research/projects/iv4xr/aplib/apidocs/
The key class is PrologReasoner, in the package nl.uu.cs.aplib.agents. The class provides APIs to add facts and rules to the Prolog engine to form a model, and how to query the model.
A tutorial on using the Prolog engine is linked from this documentation on agent programming in the Framework-core:
https://github.com/iv4xr-project/aplib/blob/master/docs/agentprogramming.md

Online solver: the first algorithm and the results of our first study can be found in this paper:

Using an Agent-based Approach for Robust Automated Testing of Computer Games, Samira Shirzadehhajimahmood, Wishnu Prasetya, Frank Dignum, Mehdi Dastani and Gabriele Keller. In the 12th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation, co-located at the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), 2021. Pdf: https://zenodo.org/record/5208858#.YR_3UI4zaiM
The iv4XR plugin for defining Reinforcement Learning environments with the SUT is available in the github repository: https://github.com/iv4xr-project/iv4xr-rl-env
The Python connector for DRL Agents and the implementation of the TD3 algorithm as a goal solver are available in the github repository: https://github.com/iv4xr-project/iv4xr-rl-trainer
Details about the approach and illustrations of usage are available in both projects' README and Wiki

</td></tr>
<tr><td>Plan for the next year</td><td>

Implementing model-based goal solvers.
Developing a more advanced online algorithm that can also construct models.
Re-implementing the online solvers from their separate projects into the Framework.
Increasing the performance of the current RL Agent and transitioning to MAEV's final target scenario.

</td></tr>
</table>

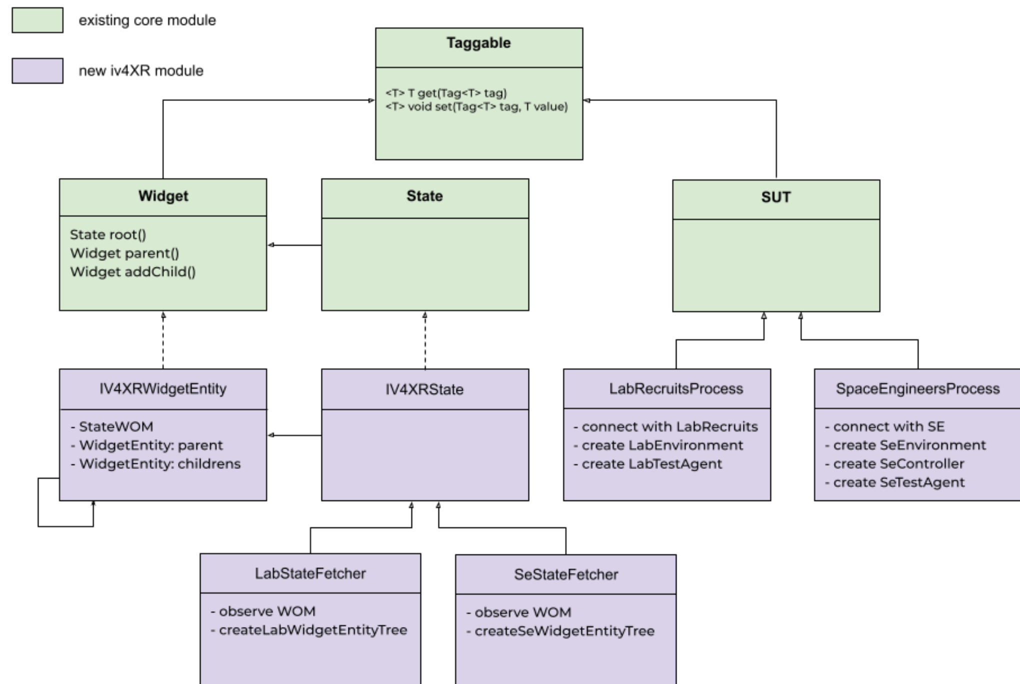
TASK 3.2 : EXPLORATION AGENTS

Short introduction to the task	The second type of FTAs focus on *exploring* the virtual environment. Like Goal Solving FTAs, exploratory FTAs observe the WOM to obtain the information of the reachable entities. However, the decisions on what to test are not based on following specific instructions, instead, the goal is to explore. Exploratory FTAs perform a non-sequential exploration of the environment, and verify the behavior of the system while trying to learn potentially interesting actions. Exploratory FTAs in iv4XR are based on the **TESTAR** tool that existed before the project and was a result the FP7 project FITTEST[5]. An overview paper[6] of the current state of the art of the tool has been published in the STVR journal in 2021. Although the uses for VR/XR systems as investigated in this project have not been described yet in the paper, we did add acknowledgement to the iv4XR project to the paper (together with many other projects and people). That is because several revisions of the paper have been made during iv4XR project execution. TESTAR[7] is an open source tool for automated testing using a scriptless approach. The underlying principle of TESTAR is: generate test sequences of (state,action)-pairs by connecting to the System Under Test (SUT) in its initial state and continuously select an action to bring the SUT in another state and check oracles (see Figure T1). Figure T1: TESTAR operational flow

[5] https://cordis.europa.eu/project/id/257574

[6] https://onlinelibrary.wiley.com/doi/full/10.1002/stvr.1771

[7] TESTAR, Test your system from the GUI: https://testar.org/

What has been done until the second year: **results**	**1.** The iv4XR plugins have been integrated into TESTAR allowing the tool to become an exploratory FTA for iv4XR SUTs. Plugins that have been integrated until the delivery of this deliverable are: WOM for LabRecruits and the WOM for SE systems. The new iv4XR module allows TESTAR to detect the state (TESTAR step 1) by extracting the information of the existing XR/VR entities. Figure T2 shows the new implementation, LabRecruitsProcess and SpaceEngineersProcess classes contain the functionality to launch the corresponding SUTs respectively, create the environment that allows to obtain the WOM and execute interaction commands, and create a test agent to be able to execute goal actions. Then LabStateFetcher and SeStateFetcher use the environment to observe the WOM and create the State and the Widget-Entity tree.  Figure T2: TESTAR iv4XR state module In order to derive actions and interact with XR/VR systems (TESTAR step 2), the new iv4XR module also defines two new types of actions, something that we can observe in Figure T3. An actionCommand is the simplest interaction that the SUT plugin allows to perform (eg. move), and the actionGoal are the SUT specific tactics and goals that a test agent can execute (eg. navigate to entity).

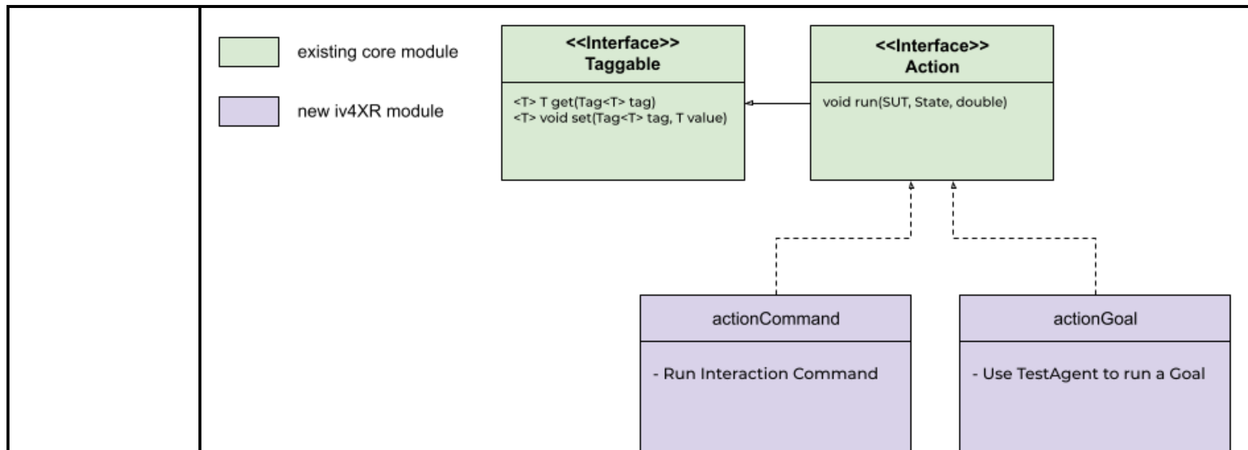


Figure T3 : TESTAR iv4XR action module

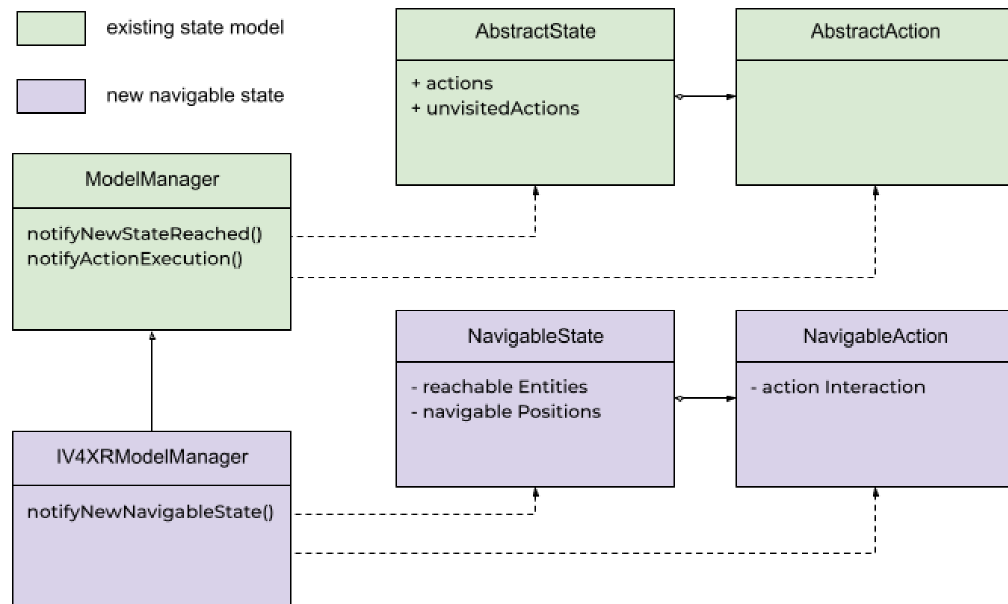
2. To enhance step 3 of TESTAR, the action selection, we have implemented a first version of a Reinforcement Learning (RL) framework. This RL uses the information of the existing states and executed actions to calculate and assign rewards values and learn from them to better select actions. Rewards that have been implemented (but not yet evaluated) are based on the heuristic that actions that cause a lot of state change lead to good testing.

We are currently investigating ideas to be applied on XR systems to assign rewards for better action selection, e.g. for LabRecruits is interesting to reward the states that contain interactive buttons. And we want to also investigate the RL approach with free exploration strategies for those XR systems that do not offer a default navigation map.

3. TESTAR integrates, by default, generic oracles (TESTAR step 6) that allow it to verify the robustness of the system. These oracles allow TESTAR to detect if the SUT process has crashed or hung, or if the SUT state, process or external log file (as in the case of SE) contains an exception message.

4. TESTAR is capable of inferring a state model while exploring LabRecruits and SE environments. This model allows the tool to remember what states were discovered and which actions were executed, something that helps to improve the action selection. Extensions related to oracles specific for the SUts in the project, are planned for the coming year.

TESTAR state model was originally designed for traditional Graphical User Interfaces (GUI) desktop applications. In most desktop applications, the GUI state contains all the available widgets to interact with. However, for XR systems, determining the XR state and reachable entities are based on the observation range and position of the agent, something that can change constantly with agent movements.

	For this reason, TESTAR state model is being extended with the concept of navigational state (see Figure T4). The objective of this new NavigableState is to allow TESTAR to explore the navigable positions of the environment as it saves the information of which entities are reachable, then execute an interaction action defined as NavigableAction (eg. open a door by interacting with a button), to continue with a new exploration of the available state positions.  Figure T4 : TESTAR iv4XR navigable state module
Where are these results and how to use them	1. TESTAR FTA with the results described above can be found on github: https://github.com/iv4xr-project/TESTAR_iv4xr a. RL framework: https://github.com/iv4xr-project/TESTAR_iv4xr/tree/v3.2/testar/src/nl/ou/testar/ReinforcementLearning b. Navigable State branch: https://github.com/iv4xr-project/TESTAR_iv4xr/tree/master-navigable-state c. LabRecruits instructions: https://github.com/iv4xr-project/TESTAR_iv4xr/wiki/LabRecruits-execution-instructions d. SpaceEngineers instructions: https://github.com/iv4xr-project/TESTAR_iv4xr/wiki/Space-Engineers-execution-instructions 2. Videos are here: a. TESTAR download instructions i. https://www.youtube.com/watch?v=SNUoLFrTmG0 b. TESTAR interaction with LabRecruits

	i. https://www.youtube.com/watch?v=st4FL_mMflE ii. https://www.youtube.com/watch?v=3T4v3STVMVU c. TESTAR interaction with Space Engineers i. https://www.youtube.com/watch?v=C-y-jV82K50 ii. https://www.youtube.com/watch?v=HKWsjWV0hmo 3. Published papers are: a. *Deploying TESTAR to Enable Remote Testing in an Industrial CI Pipeline: A Case-Based Evaluation*, Pastor Ricos, Fernando and Aho, Pekka and Vos, Tanja and Torres Boigues, Ismael and Calas Blasco, Ernesto and Martinez Martinez, Hector, in 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20–30, 2020, Proceedings, Part I. DOI: https://doi.org/10.1007/978-3-030-61362-4_31 *Zenodo:* https://zenodo.org/record/5111003 b. *TESTAR – scriptless testing through graphical user interface, Tanja E. J. Vos, Pekka Aho, Fernando Pastor Ricos, Olivia Rodriguez-Valdes, Ad Mulders, in Software Testing, Verification and Reliability, Volume 31, Issue 3 Special Issue:New Generations of UI Testing, May 2021, DOI:* https://doi.org/10.1002/stvr.1771
Plan for the next year	• Experiment with different RL approaches • Extend generic oracles to adapt them to XR use cases • Investigate appropriate abstraction mechanisms for XR systems • Study the complementarity of TESTAR navigational state with MBT coverage tool (Task 3.4 : Coverage)

TASK 3.3 : DEALING WITH HAZARDOUS ELEMENTS

Short introduction to the task	Hazardous entities are entities in the virtual world that may block or even sabotage an agent's progress or even cause it to fall into an inescapable stuck state. To deal with them the agent needs to be actively aware of their threat and apply countermeasures.
What has been done until the second year: **results**	• Within the FTAs that focus on SUT Goal Solving, where we use the Test Specification Language (TSL) developed in Task 3.1 (see D3.1 - "Test Specification Language"), agents can take deliberations about which strategies will allow them to navigate and interact with different entities to solve their goals.

	• Solving goals to test specific tasks, means that dealing with hazardous elements (Task 3.3) will not be very different from dealing with other XR application elements. • Consequently, the goal solving FTA can already handle these hazardous elements which completes Task 3.3 and its inclusion in the FTA prototype. This task has finished.
Where are these results and how to use them	• LabRecruits tactic library: https://github.com/iv4xr-project/iv4xrDemo/blob/master/src/main/java/agents/tactics/TacticLib.java
Plan for the next year	Active development of this task is officially completed. However, in this final year of the project, partners are expecting maintenance updates of the tactics in the iv4XR framework.

TASK 3.4 : COVERAGE

Short introduction to the task	While the basic task of solving a test goal by the FTA is handled in Task 3.2, in this task the objective is to measure the coverage of the tests to see how good they are in exercising the system under test (SUT). Furthermore, the functionality of the FTA is enhanced in order to increase their efficacy in achieving higher levels of coverage. The activities in this task are focused along two main lines: 1) define reasonable metrics for measuring coverage that is applicable to all the use cases we have in the project, and 2) develop test generation strategies that increase the coverage obtained from the tests. For the first point, different notions of coverage are explored and being explored in such a way that test generation could be geared towards achieving high coverage, while for the second point approaches based on search as well as reinforcement learning are explored. Thales SIX developed its approach on coverage with Diversity RL. The main idea is that we use as a foundation the DRL solver from T3.2, but we want to achieve behavioural coverage. Whereas in a classic RL training setup we are learning a single control policy that fulfills our goal, with Diversity RL we aim

	at learning a set of diverse and successful policies to fulfill our goal. In the Thales powerplant intrusion example, this lets us obtain different intrusion strategies due to different flaws of the defense strategy, that can thus be corrected by the SUT user.
What has been done until the second year: results	Given the nature of the SUT where the interactions with the system are fine grained and infinitely many to enumerate in a meaningful way, we have adopted notions of coverage that are at a higher level of abstraction. However, such notions of coverage are still relevant for the tester as they capture the space of interactions in the SUT. In particular, we consider notions of coverage defined on models that represent abstractions of the interactions in the SUT. Given such a model, we calculate the levels of coverage on the model, in particular state and transition coverage. State coverage measures how many of the states in the model are covered, while transition coverage measures how many of the transitions are covered. State coverage corresponds to how many of the entities in the SUT are interacted with, while transition coverage quantifies how many of the user actions have been explored. Further notions of coverage, such as path coverage, are currently being considered. Path coverage would correspond to a complete scenario in the SUT (e.g., one game play scenario in LabRecruits or Space Engineers). A tool called *MBT* (which stands for ‘model-based testing’) for the generation of test suites that achieve high coverage of the model has been developed. The MBT tool employs different search-based heuristics to generate test cases so as to cover all possible targets in a given model of the SUT. On the other hand, we are exploring reinforcement learning for the generation of test cases. In particular, for Lab Recruits we have built a basic explorative implementation that allows the FTA to explore the SUT. Similarly for the use case from Thales on Nuclear plant intrusion simulation, reinforcement learning is being applied. For Thales’ Diversity RL approach, we based our work on the QD-RL paper. QDRL maintains a population of agents that are designed to satisfy both

<table>
<tr><td></td><td>Quality characteristics, being efficient at solving the task at hand, and Diversity characteristics, proposing agents that display a variety of behaviors. QDRL is an iterative algorithm that selects the most promising agents in the current population and mutates them to create new agents. The most promising agents are the one in the Pareto front in the two dimensional space where agents are represented by their performance and diversity pair. Half of the selected agents are mutated so as to maximize their quality while the other half optimizes its diversity score. This mutation is done using state of the art DRL techniques namely using the update criterion of TD3. Note that making TD3 work in the maze environment was carried on successfully as part of Task 3.2.

Our implementation of the QDRL paper differs from the original version as the objectives are slightly different. While the objective of vanilla QDRL is to find one optimal strategy by exploring diverse policies, our objective is to output as many diverse optimal policies as possible. This motivates why, when comparing the trajectories generated by the policy to determine diversity scores, our version compares the whole trajectories instead of just the final state (position) of the agent, therefore comparing the behaviours to navigate the maze instead only what the policies were able to achieve.

Our current implementation of QDRL is capable of outputting multiple successful policies in a simple maze environment without guards.</td></tr>
<tr><td>Where are these results (paper, github, etc)</td><td>The combined application of model-based and search-based test generation for maximizing coverage has been presented at the 13th Symposium on Search-Based Software Engineering (SSBSE). The data and replication material accompanying the publication are available in the project Zenodo repository: https://zenodo.org/record/5140432
The MBT tool source code is available in the project github repository, together with all the necessary resources and documentation to execute it: https://github.com/iv4xr-project/iv4xr-mbt
The implementation of reinforcement learning for test generation applied to Lab Recruits is also available in the project github repository: https://github.com/iv4xr-project/iv4XR-FTA-RL
The implementation of the QD-RL algorithm for behavioural coverage is available in the github repository: https://github.com/iv4xr-project/iv4xr-rl-trainer
Details about the approach and illustrations of usage are available in the project README and Wiki</td></tr>
<tr><td>How can they evaluate/use</td><td>Usage instructions for the MBT tool are detailed in the readme page: https://github.com/iv4xr-project/iv4xr-mbt/blob/master/README.md</td></tr>
</table>

these results (what is there and what should they do to evaluate or use it)	Moreover, the MBT tool provides help on the various options available at runtime, suffices to simply run the tool without any parameters and a help page should be displayed. ● MBT performs: ○ Coverage driven abstract test generation from a given EFSM model ○ Concretization of the abstract tests and execution on the system under test (Lab Recruits) ○ Mutation analysis of the generated tests to assess their fault finding potential. ● Experimental results are available from our zenodo repository: https://zenodo.org/record/4769901
What is the plan	● Enhancement to the MBT tool ● Finalize RL-based coverage driven test generation and perform evaluation ● Evaluate coverage effectiveness on iv4XR use cases, beyond Lab Recruits ● Add support for multi agent environments ● Scale up the results for QDRL: progress to the maze environment with guards, then the power plant environment. Evaluate different diversity metrics and adapt it to consider guard detection events when the guards behavior is not fully deterministic

TASK 3.5 : MULTI AGENT TESTING

Short introduction to the task	Many XR systems allow the simultaneous interaction of multiple users in the same environment. This implies the need to verify the correct interaction of multiple users, since they can influence each other. This task is focused on extending the iv4XR framework to allow the communication of multiple agents in runtime. There are two main objectives for this task: - Allow the definition of test cases that involve simultaneous interactions, collaboration or confrontation of multiple agents. - Improve entity-search and exploration performance by coordinating a group of agents to achieve a common goal.
What has been done until the second year: **results**	In order to allow the communication of multiple agents, the iv4XR framework introduces a concept called *communication node*. Agents can register to such a node, which then facilitates communication between these agents. An agent can then send a message to the node to communicate a desired event. The message can be sent to a specific agent, a group of agents (identified by a common role) or broadcasted to all agents registered to the same node.

	The example below shows how two agents are registered to a node:

```
ComNode communication = new ComNode();
var agentOne  =  new LabRecruitsTestAgent("AgentOne","")
. attachState(new BeliefState())
. attachEnvironment(env)
. registerTo(communication);
var agentTwo = new LabRecruitsTestAgent("AgentTwo","")
. attachState(new BeliefState())
. attachEnvironment(env)
. registerTo(communication);
```

To provide the functionality to the agents to decide whether to send a message or wait to receive them, the tactical-goals implementation has been extended as well as the TSL language that allows users to construct test cases. The example below shows how an agent can send a message to another specific agent. And how the other agent is able to receive the information of this message:

```
agentOne.getState().messenger()
.send("AgentOne", 0, MsgCastType.SINGLECAST, "AgentTwo", "blabla");

agentTwo.getState().messenger()
.retrieve(M -> M.getMsgName().equals("blabla"));
```

For LabRecruits, we have used a multi-agents implementation together with machine learning techniques to create both collaborative and non-collaborative pairs of socio-emotional test agents (SETAs). These SETAs were then used by WP4 to explore ways of measuring collaboration between agents or real users, and to propose a method of evaluating whether a scenario is collaborative or not.

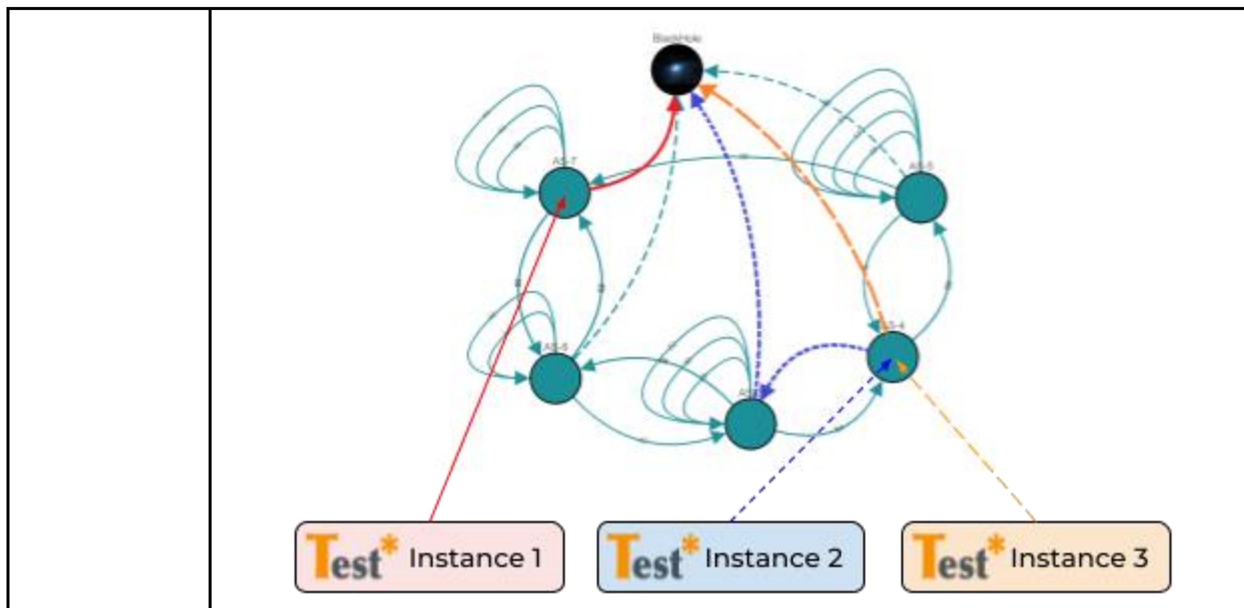
Use cases:

As a multiplayer game, Space Engineers requires occasional “Mass testing” wherein a large group of players from the closed beta test group connect to a single server and play. The purpose of this is to check the stability of the server and underlying engine. These tests require some coordination, however, in order to get enough numbers and also require some preparation in terms of measurements to take and recording infrastructure to maximise the opportunity.

The current plugin can emulate the actions of a single player using an SE client connected to the game server. However, this is not all that efficient as the game has been designed with the assumption of one-player-per-client, which would mean that *n* automated clients would need *n* copies of the game and subsequently *n* running instances of the framework.

	Work on this front has been in allowing the game to spawn multiple “NPC” engineers and controlling them through the same plugin API. This way, we can host multiple agents on the same instance of the game and process their actions. Because of the above assumption, these NPC agents are not as capable as the “player agent”, so the current focus on the multi-agent side is in emulating the actions of NPC agents so that they can perform the same actions as the player agent. ***Thales-Six*** With its Diversity RL approach, Thales trains multiple agents, but each agent interacts in its own environment. We either use the “reset” capability of the environment to have successively agent 1, agent 2, …, agent N act with the environment; or we use parallel instances of SUT and iv4xr to have one environment instance by agent. For now the first approach with the “reset” function is implemented. In their training procedure, the agents share a common interaction memory, referred as Replay Buffer in the Deep Reinforcement Learning domain, that is exploited for their model updates. The adapted QD-RL algorithm also manages the evolution of the agent population to have them explore the state space efficiently, thanks to the diversity criterion that emphasizes the difference between two agents’ trajectories. ***GWE*** Many engineering monitoring projects are so complex that the site is broken down into sub-sites, each of which have their own hosting server which connects to the sensors for that area. We used the same hierarchical approach for these types of complex sites, to trigger initial multiple agents for each sub-site, each of which can then further analyse the sensors at that sub-site. Projects containing a single site/location are also processed by multiple agents as our server-based structure already caters for multi-threaded access to the large volumes of sensor data. Our JAVA tool can detect simple sensor errors such as flatlines, jitter, missing values, calibration errors, as well as more complex errors such as formulae-based readings being invalid due to one or more errors in any sensor which is used by the formulae. When an initial agent discovers errors or potential errors,

	further agents are created to analyse those parts of the data or sensor parameters in further detail, using a separate process.
Where are these results and how to use them	• Aplib multi agent package: https://github.com/iv4xr-project/aplib/tree/1.4.0/src/main/java/nl/uu/cs/aplib/multiAgentSupport • Aplib multi agent tutorial: https://github.com/iv4xr-project/aplib/blob/master/docs/manual/tutorial_3.md#multi-agent • SE: https://github.com/iv4xr-project/iv4xr-se-plugin/blob/v0.4.0/JvmClient/README.md#multiple-characters • Usage of SETAs to evaluate whether a LabRecruits scenario is collaborative or not. *“Collaboration analysis in multi-player based simulations”* https://fenix.tecnico.ulisboa.pt/cursos/meic-t/dissertacao/846778572212536
Plan for the next year	• Add multi-agent support into the MBT tool for coverage (Task 3.4 : Coverage) • SE will implement multi-agent test cases • Thales-Six will train and experiment with Diversity multi-agent RL approach • TESTAR distributed state model inference: Because an in-depth exploration of a virtual environment requires the execution of a large number of actions, we want to divide and coordinate the execution of these actions among multiple TESTAR instances to speed up the exploratory process. In order to do that, we are developing a framework in the TESTAR tool that allows the inference of a state model in a distributed way. Multiple TESTAR instances coordinate their action selection while exploring the same SUT by using shared knowledge of a state model that remains in a centralized database.

Test* Instance 1
Test* Instance 2
Test* Instance 3