Intelligent Verification/Validation for XR Based Systems

Research and Innovation Action

Grant agreement no.: 856716

D3.5 – Report describing Functional Test Agents (FTAs)

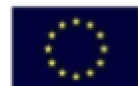
iv4XR – WP3 – D3.5

Version 1.10

December 2022

Project Reference	EU H2020-ICT-2018-3 - 856716
Due Date	31/12/2022
Actual Date	30/12/2022
Document Author/s	Tanja Vos (UPV), Fernando Pastor Ricós (UPV), Borja Davó Gelardo (UPV), Wishnu Prasetya (UU), Fitsum Kifetew (FBK), Davide Prandi (FBK), Raihana Ferdous (FBK), Victor Gabillon (THA-SIX), Joseph Davidson (GA), Pedro Fernandes (INESC-ID), Inês Carvalho (INESC-ID), Rui Prada (INESC-ID), Manuel Lopes (INESC-ID), Jeremy Cooke (GWE)
Version	1.10
Dissemination level	Public
Status	Final
Type	REPORT

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	20/11/2022	Initial document structure and contents	FP
1.1	29/11/2022	Discuss deliverable structure	TV, WP, FK, FP, VG, RP, PF, JC, JD, DP
1.2	5/12/2022	Add Exploratory FTA section	FP
1.3	9/12/2022	Add INESC-ID quality-diversity to Coverage section	PF
1.4	9/12/2022	Add RL section introduction and Quality-Diversity approach	VG
1.5	13/12/2022	Add Goal Solving section	WP
1.6	14/12/2022	Add Augmented Reality section	BD, IC, RP
1.7	15/12/222	Add RLbT and EvoBMT sections	DP, FK
1.8	19/12/2022	Update RLbT section	RF
1.9	20/12/2022	Rewrite some text in the AR Testing section	RP
1.10	30/12/2022	Final arrangements for submission	RP

Document Quality Control			
Version QA	**Date**	**Comments (and if appropriate reason for change)**	**Initials of QA Person**
1.8	19/12/2022	Minor comments and edits	RP

1.9	20/12/2022	Overall revision	ML
1.9	28/12/2022	English grammar, sentence structure, figures	JD

Document Authors and Quality Assurance Checks

Author Initials	Name of Author	Institution
TV	Tanja Vos	UPV
FP	Fernando Pastor	UPV
BD	Borja Davó	UPV
WP	Wishnu Prasetya	UU
FK	Fitsum Kifetew	FBK
DP	Davide Prandi	FBK
RF	Raihana Ferdous	FBK
VG	Victor Gabillon	THA-SIX
JD	Joseph Davidson	GA
JC	Jeremy Cooke	GWE
RP	Rui Prada	INESC-ID
PF	Pedro Fernandes	INESC-ID
IC	Inês Carvalho	INESC-ID
ML	Manuel Lopes	INESC-ID

TABLE OF CONTENTS

Executive Summary 1
Acronyms and Abbreviations 1
Overall concepts, architecture, design Functional Test Agents (FTAs) 2
1. Goal Solving FTA 3
1.1 Offline goal solving with no hidden transition 5
1.2 Offline goal solving on models with extended state structures 5
1.3 Online goal solving through tactics 8
1.4 Online goal solving with constrained observability 10
2. Reinforcement Learning FTA 14
2.1. Introduction 14
2.2. Reinforcement Learning Based Testing (RLbT) 15
2.3. Quality-Diversity RL (Thales MAEV) 19
3. Scriptless Exploratory FTA 22
3.1. Introduction 22
3.2. TESTAR iv4XR extension 23
3.3. TESTAR Output Files 30
3.4. References for documentation, videos and papers 31
4. Coverage 31
4.1. EvoMBT: Evolutionary Model Based Testing 32
4.1.1 Model-based testing 32
4.1.2 EvoMBT software architecture 34
4.1.3 Case Studies 35
4.2. Quality-Diversity Optimisation for Testing Space Engineers 38
5. Augmented Reality 40
5.1. Introduction 40
5.2. Use Case 40
5.3. The testing process 42
5.4. Integration with the iv4XR framework 46
5.5. AR Tests 47
FTAs Scientific Publications 52
References 54

EXECUTIVE SUMMARY

This deliverable D3.5 is a report that describes the existing Functional Test Agents (FTAs) of the iv4XR framework and their technical approaches for testing the XR use cases. The project's GitHub repository contains extended and updated documentation regarding the specification of each FTAs.

The document starts with an overall description of the integration of the FTAs within the iv4xr framework. Then it continues as follows:

1. **Goal Solving** agents that follow goal instructions to verify the XR functionality.
2. **Reinforcement Learning** agents that learn gradually from the environment to solve tasks.
3. **Exploratory** agents that execute non-sequential actions to verify the system's robustness.
4. **Coverage** agents that use a model crafted by a domain expert to generate test cases.
5. **Augmented Reality** agents that follow goal instructions to verify the AR scenarios.

ACRONYMS AND ABBREVIATIONS

FTA	Functional Test Agent
XR	eXtended Reality
VR	Virtual Reality
AR	Augmented Reality
RL	Reinforcement Learning
QDRL	Quality-Diversity Reinforcement Learning
MBT	Model Based Testing
SUT	System Under Test
WOM	World Object Model
GUI	Graphical User Interface
SE	Space Engineers
DFS	Depth First Search
EFSM	Extended Finite State Machine
LTL	Linear Temporal Logic

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

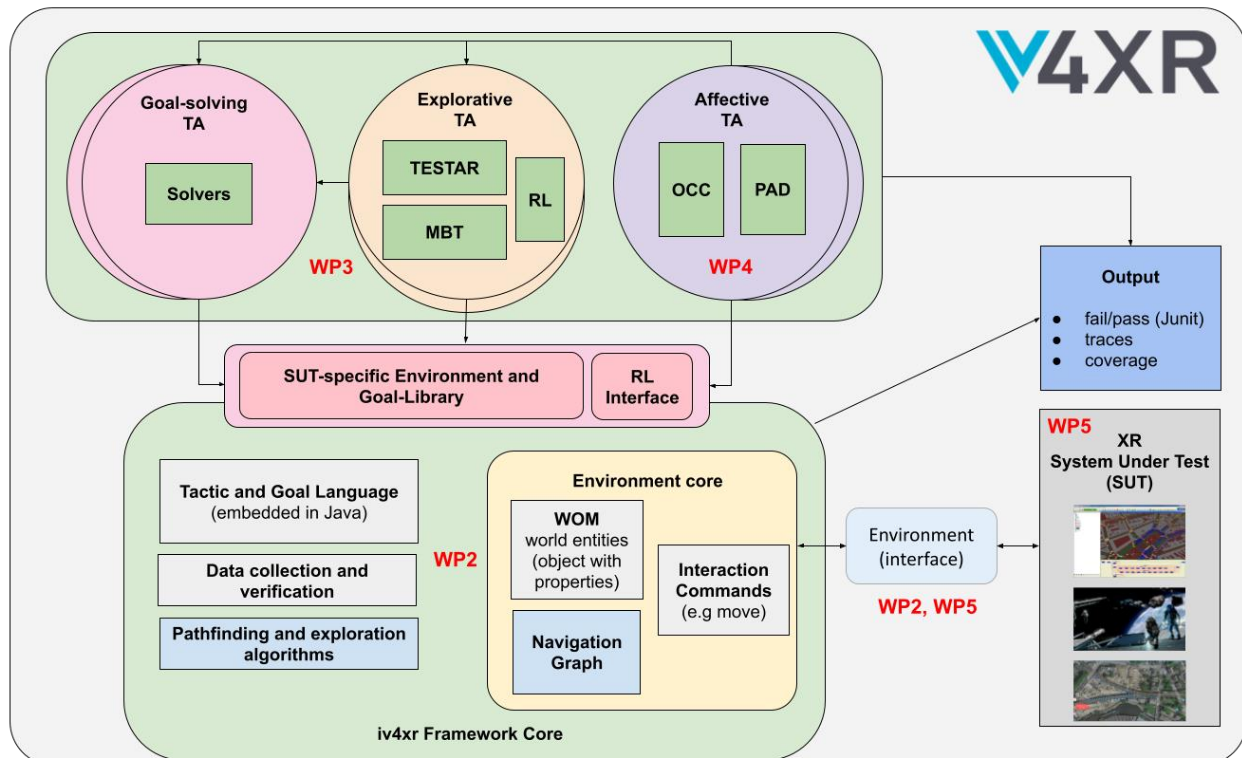


Figure 1: Different types of FTAs in iv4XR.

We distinguish between four sub-types of FTAs in WP3, of which three of them follow similar exploratory capabilities:

- The first type of agent makes deliberations to choose the appropriate strategies that will allow it to solve goals(**Goal-solving** TA circle in **Figure 1**).
- The second type of agent does not follow specific goal structures but learns from the executed goal interactions to verify if it is possible to achieve a final state (**RL** section from **Explorative** TA circle in **Figure 1**).
- The third type of agent explores the XR environment with a *Scriptless* approach while verifying if the system is robust enough to respond to multiple and unexpected user interactions (**TESTAR** section from **Explorative** TA circle in **Figure 1**).
- The fourth type of agent can follow the space of interactions that are abstractly represented in a model to cover all the transitions (**MBT** section from **Explorative** TA circle in **Figure 1**).

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

1. GOAL SOLVING FTA

This section will present a number of goal solving algorithms available to iv4xr functional test agents. They are used to target different classes of problems, but all are relevant for XR setups. To allow them to be discussed in a similar way, let us first introduce a generic setup for goal solving problems. The system under test (SUT) will be abstractly *modeled* as a Finite State Machine (FSM) $M=(\Sigma,A,T,s_0)$ where Σ is M's set of possible states (finite), A is a set of labels representing names of actions, T is a set of transitions between states, and $s_0 \in \Sigma$ is the initial state (for simplicity we will assume just a single initial state). A transition is a triple (s,a,t) where s is the origin state of the transition, t is the transition's result state, and a is a label from A.

The states in M can be seen as *abstractly* representing SUT's actual states. A possible testing concern is, for example, to check the consistency of the actual state that is represented by some abstract state s, presuming that the testing program is given some methods to inspect the actual state. Note that for the same SUT, we may have multiple models, e.g. providing different perspectives on the SUT. For example we can have a model M_1 that captures the SUT's functional logic, and a model M_2 that captures navigation between different key locations in the SUT's virtual world.

A test agent can be seen as a program that drives M to move from one state to another by executing its transitions, starting from the initial state. E.g. the agent may want to visit a particular state s in M to check the consistency of the actual state that s represents, or to check that all outgoing transitions of s can indeed be executed.

We will also distinguish between a complete model of the SUT, and the model that the agent knows. Imagine a model *M0* that we can regard as the complete model of the SUT. By "complete" we do not mean that it captures every aspect possible of the SUT (such a model will not be a 'model' anymore); but rather a model that with respect to some chosen abstraction can be regarded as complete. It is not necessary that we actually have this *M0*, it is just a useful concept to introduce. An agent can be given a partial model M, e.g. because constructing the complete *M0* takes too much work. An agent can also come with an ability to learn *M0*. That is, it can be given some very limited M at the beginning, but as it interacts with the SUT it gradually extends M.

Def.1.1: A model $M_1=(\Sigma_1,A,T_1,t_0)$ is as *submodel* of $M_2=(\Sigma_2,A,T_2,s_0)$ if (1) the initial states are the same, (2) $\Sigma_1 \subseteq \Sigma_2$, and (3) any *non-hidden* transition of T_1 is also a transition in T_2.

Def.1.2: A model is said to have *hidden transitions* if it contains transitions of the form $(s,a,?)$ where "?" is a symbol to mean unknown. This means that there is a transition labeled with a available on the state s, but it is unknown where this transition leads to.

We can distinguish several testing setups:

- **Full model** setup: the agent is given a complete model *M0*. This setup is not always possible or feasible. In some situations, *M0* can be extracted automatically from the SUT. If there is no such mechanism, then it has to be hand crafted, which can be expensive.
- **Partial model** setup: the agent only knows a part of *M0*. The knowledge can be partial in various ways. E.g., it may know some sub-model *M* of *M0*. In a different setup, the agent might know which transitions are available at each state, but it does not know where those transitions would lead until it executes them (in other words, *M* has hidden transitions).
- **Online** testing setup. In this setup, transitions are executed on the actual SUT. The benefit of this is that this allows us to check the SUT's actual states. The drawback is that online execution of a transition is slow. Since many combinations have to be tried, overall such an approach is computationally expensive. If the agent already knows which sequence of transitions it wants to do, the cost is usually acceptable. Pre-planning the transitions works in the full model setup, but may not be possible in a partial model setup, in particular if it has many hidden transitions. In the latter case, it may become necessary for the agent to try out different transitions online in order to figure out how to get to a certain state. The computation cost of this can be excessive.
- **Offline** testing setup. The agent executes the transitions on *M* without executing them on the SUT. This still allows us to check some correctness properties, e.g. to check if a certain predicate φ over Σ is reachable. The more important benefit we get from this setup is the ability to plan, e.g. to find a sequence of transitions that ends in some desired state, or to find a sequence of transitions that would cover some transitions, or a pair of transitions. Importantly, the solution is calculated offline! In the full model setup, this would allow us to generate a test suite, consisting of sequences of transitions as test cases, that would cover e.g. all transitions in *M0*, or all pairs of transitions in *M0*. Since offline executions do not require executions on the SUT, offline planning is computationally cheap (fast). Once obtained, the sequences can be executed online on the SUT to do actual testing. While the online testing part would still incur its computation cost, the more complex calculation to search for a covering test suite is done offline, which would save us a lot of computation cost.

Imagine that the agent knows a model $M=(\Sigma,A,T,s_0)$ of the SUT. This *M* is not necessarily the full model *M0*. A *goal* is represented by a predicate φ over Σ. In an agent-based setup, a goal represents states that we want the agent to be. In terms of testing, these could be states that the agent needs to check for their consistency.

Def.1.3: A goal φ is *solved* from a current state *s* if we can find a sequence σ of transitions in *T* that would bring *M* from the state *s* to a state *s'* such that $s' \models \varphi$ (φ is true on *s*'). Such a σ is called a solution of φ.

A goal solving algorithm is simply an algorithm for finding a solution for a given goal.

1.1 OFFLINE GOAL SOLVING WITH NO HIDDEN TRANSITION

In this setup, *M* may be partial, but it does not have any hidden transitions. This setup is pretty straightforward to solve, e.g. we can just apply an offline depth first search (DFS) on *M* to find a state satisfying goal φ. Since there is no hidden transition, we can also solve φ completely offline. Of course, we can then only solve φ in this way if it is solvable with the information we have on *M*.

In the case that φ specifies a singleton state $\{t\}$, the problem is the same as the pathfinding problem over a graph, namely to find a path from the current agent state *s* to *t*. For this iv4xr provides an implementation of the A* algorithm, which is efficient and often gives a better (shorter/shortest) path than DFS, provided a concept of distance between nodes/states is given. A typical setup where this is used is when *M* is actually a navigation graph *NG* over a virtual world. Every *s* in Σ represents a visitable location in the virtual world. A transition between two states *s* and *t* means that the agent can travel (in the virtual world) in a straight line from *s* to *t* without encountering any obstacle (e.g. there is no tree in between that can block this travel). The distance between them can be defined as the physical straight line distance between them. Some SUT can produce a complete navigation graph *NG0* which the agent can exploit. Else iv4xr provides a method that can construct a navigation graph on the fly (as the agent explores the world).

List<NodeId> ***findPath***(*Navigatable NG, NodeId start, NodeId goal*)

Figure 1.1: *the API for invoking A* pathfinder.*

1.2 OFFLINE GOAL SOLVING ON MODELS WITH EXTENDED STATE STRUCTURES

Consider now a setup where *M* is extended with a set *V* of variables. Transitions can be guarded by a condition/predicate over *V* and can update the values of *V* as well. The domain of V (the values that the variables can take) does not need to be finite. Such an *M* is also called *extended finite state machine* (EFSM) [AP11]. The states in Σ is then called the '*abstract states*' of *M*. If full state is a pair of (*s,v*) where *s* is an element of Σ and v is a vector of the current value of the variables in *V*; it is also called *configuration*.

Note the space of possible configurations of an EFSM can be *infinite*. EFSM is also Turing complete.

Consider a more expressive formulation of the goal, namely using a Linear Temporal Logic (LTL) [BK08] formula ψ. Such a formula is a predicate over infinite sequences of states, rather than simply a state predicate. Iv4xr supports LTL. A fragment of supported syntax is shown below; a more complete description is presented in the D2.4 report. Below, *p* is a state predicate as an atom, ψ is an LTL formula, and *F* is either a state predicate or an LTL formula.

F ::= p | ψ
ψ ::= **now**(p)
| **next**(F) -- also known as the X operator
| **always**(F) -- also known as the □ operator
| **eventually**(F) -- also known as the ◇operator
| **ltlAnd**($\psi_0, \ldots, \psi_{n-1}$) -- conjunction ∧
| **ltlNot**(ψ) -- negation ¬

The semantics of LTL formulas over infinite sequences is defined as usual [BK08][1]. Traditionally LTL is used for verification. That is, ψ is used to express a correctness requirement, and we would then want to know if all possible executions of M satisfy ψ. Here, we want to use ψ as an expression of goal, e.g., it might be used to encode a certain test scenario (e.g. we want to have a run that first visit room-1 and then room-2, and so on). To simplify the discussion, we will only consider a setup where a solution is wanted with respect to *M*'s initial state s_0.

Def.1.4: given an LTL goal ψ, a solution to this goal is a "*finite witness*" of ψ.

Def.1.5: An *execution* of *M* is a sequence π of configurations starting in M's initial configuration, and furthermore for any pair of consecutive configurations in π_i and π_{i+1}, there exists a transition in *M* that can be executed on π_i and would result in π_{i+1}.

Def.1.6: A *finite* execution π is a *finite witness* for an LTL formula ψ if either: (1) any infinite execution that extends it would satisfy ψ as well, or (2) π contains a suffix that is cyclic, that when repeated indefinitely would yield an infinite execution that satisfies ψ.

Defined as above, a solution for ψ can be obtained by applying an LTL model checking algorithm e.g. as in [BK08], provided the space of possible configurations of *M* is still finite. In iv4xr, an implementation of LTL model checking is provided[2]. The iv4xr LTL model checker uses a double DFS approach similar to what is used in the SPIN model checker [Ben08]. The iv4xr LTL model checker actually applies bounded model checking (BMC), so it will actually work on *M* with an infinite space of configurations, though at the expense of losing the completeness guarantee. It can also give the shortest solution, by applying a binary search over the depth bound.

It can be noted that the choice for LTL, rather than other modal logics such as CTL, has been deliberate. LTL, being a sequence predicate, gives a natural concept of executable solution (to goal solving). As defined in Def. 1.4, the solution of an LTL goal ψ is a sequence of actions, which

[1] LTL is also discussed in D2.4, but note in D2.4 LTL is used to check the runs of test cases. These runs are finite, so there we need an LTL semantic over finite sequences. Here, we use LTL to specify a goal. In this setup the standard semantic over infinite runs gives us more expressiveness while retaining the ability to solve such a goal.

[2] The decision to implement our own model checker, rather than using an existing model checker, is mainly to allow easier and deeper integration with other modules within iv4xr and retaining the option to adjust or extend the checker as needed. As far as we know, iv4xr LTL model checker is the only LTL model checker that is implemented in Java. There are existing LTL model checkers, such as SPIN, or LTLmin. But SPIN, for example, does not offer APIs, and is limited to finite state space use cases, which is not necessarily our case in iv4xr. LTLmin has APIs, but is less suitable for handling states with possibly deep structures.

can be turned into a real test execution (of the action-sequence) on the SUT. Also, there is a standard procedure to find a solution, namely through model checking as pointed out above.

Similar to SPIN, the iv4xr MC is lazy, which means that *M* does not have to be literally a graph-like structure with explicit nodes and arrows. In fact, *M* can be implemented succinctly as a Java program, as long as it implements a certain interface (among other things, it requires the program to implement state cloning).

In contrast to the navigation graph mentioned in Section 1.1, typically an EFSM model is used to also capture the functional logic of the SUT. Once we obtain a solution to a goal, we typically would want to execute it on the real SUT (e.g. as a test case). The actions from the EFSM are typically high level actions which may not be directly executable in the SUT. So, some adapters may be needed to translate them to primitive actions in the SUT. In our studies we typically use a fixed set of goal structures as our collection of possible actions (the A component). As a goal structure is executable (through a test agent), we then have the ability to execute a solution.

Figure 1.2 below shows the main APIs of iv4xr bounded model checker (BMC). The constructor constructs a checker from a given model *M* of type/interface ITargetModel. The latter means that a "model" can be any Java program as long as it implements the methods of the interface ITargetModel. **Figure 1.3** shows key methods of ITargetModel. Importantly, getCurrentState() needs to return the current model's state, represented as an instance of IExplorableState. The latter requires that model states must be cloneable.

The key methods that have to be implemented by a model are shown in Figure 1.3.

```
BuchiModelChecker(ITargetModel model)  // constructor
Path<Pair<IExplorableState,String>> find(LTL ψ, int maxDepth)
Path<Pair<IExplorableState,String>> findShortest(LTL ψ, int maxDepth)
```

Figure 1.2: *the main APIs for constructing and invoking iv4xr LTL bounded model checker.*

```
interface ITargetModel :
        IExplorableState getCurrentState()
        boolean backTrackToPreviousState()
        List<ITransition> availableTransitions()
        void execute(ITransition tr) ;

interface IExplorableState :
IExplorableState clone() ;
```

Figure 1.3: *methods that should be implemented by a model, so that it can be targeted by iv4xr BMC.*

1.3 ONLINE GOAL SOLVING THROUGH TACTICS

In many setups the agent does not actually have a model to help it. A navigation graph may be available, or can be constructed on the fly, but an EFSM that models the functional logic of the SUT is much harder to come by. If reaching a goal state *s* is reachable through obstacle free travel, then having a navigation graph is enough to solve this goal offline. But more often than not, reaching a goal state requires that some logic in the SUT needs to be engaged. Without a model that captures the logic, we will then have to solve such a goal in an online way. In iv4xr we can define so-called *tactics* for solving goals. Essentially, a tactic is a set of guarded actions that define a heuristic for solving a goal.

An iv4xr agent runs a loop; each iteration is called a *deliberation cycle*. During such a cycle the agent observes the SUT and then decides which action towards reaching a goal that is given to it. The interface to the SUT (the *Environment* component in Figure 1) should provide a set of primitive methods for controlling the SUT; these are the basic actions the agent can do. However a goal may require a run of many cycles to even get close to it, so choosing the right action at every cycle is not trivial. A *tactic* is essentially a logical statement that controls how the agent makes this choice.

From a goal's perspective, a tactic is an online *solver*, as it drives the agent to do a whole series of actions that eventually solves the goal. Obviously it is not possible to write a tactic that can solve all goals. Instead, a tactic is usually written as a solver for a certain family of goals. The concept of "goal family" can be captured by a *parameterized goal*. For example, let's assume that the goal to reach an object in the same room can be solved by a general heuristic regardless of which specific object we try to reach. We can capture this by a goal e.g. *reached*(*e*) that is *parameterized* by the specific object *e* that we target. This can be solved by the same tactic that implements the heuristic, that can then solve *reached*(*e*) for any *e*, as long as it is located in the same room as the agent.

It is usually easy to write tactics for solving simple goals. For a goal that is harder to solve, it is also possible to define a *goal structure*; with it we can introduce e.g. a sequence of subgoals to help the agent in solving its main goal. Such a construct is used when it is hard for the agent to come up with the subgoals by itself, so a human gives the subgoals. However, each subgoal has a known tactic to solve it.

More about the tactic and goal structures of iv4xr agents are explained in Report **D2.4**.

Let's call a goal, which is not a goal structure, a *basic goal*. Such a goal has a tactic associated with it. With a *goal structure* we typically refer to a composition of basic goals and other goal structures. Having a library of basic goals, and a way to compose them allow us to treat these basic goals as *high level transitions* over the SUT state. That is, giving an agent a basic goal, and then running it to solve the goal, amounts to transitioning the SUT from one state (the one before the agent gets the goal) to another (the one after the goal is accomplished). Giving the agent a sequence of these basic goals amounts to executing a sequence of these high level transitions, to move the SUT to some new state.

As an example, **Figure 1.4** below shows a level design in a 3D maze-puzzle game called Lab Recruits. It has three doors *d1..d3* and four buttons *b1..b4*. The starting position of the player is indicated by the yellow circle. E.g. if we want to verify that door *d3* can indeed be opened, the logic to do this is not so trivial. Doors can be opened (and closed) by toggling the right buttons. Opening *d3* requires toggling *b3*, but access to it is guarded by a closed door *d1*. Moreover, toggling *b3* would close *d1* again.

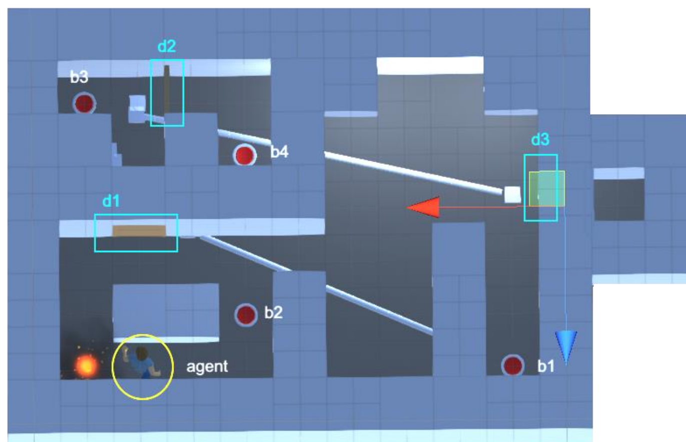


Figure 1.4: *a screenshot from a level design in a 3D maze-puzzle game called Lab Recruits*[3].

For this example game we have implemented a *library of basic goals*, it includes the following (notice that they are *parameterized* goals):

- *interacted*(*e*) : is accomplished when the entity *e* is interacted with. This goal requires the agent to be located near *e*.
- *closeBy*(*e*) : accomplished when the agent is located near *e*. If started in a location which is not near *e*, the tactic of this goal will guide the agent to *e*, provided there is an unblocked path to it. The game provides a navigation graph, so we can use it to do path planning. Importantly, note that "guiding" the agent would typically require multiple deliberation cycles to complete, during which multiple calls to primitive move-action have to be invoked.

Imagine a testing task to verify that the door *d3* can be opened. In terms of a goal predicate φ_{d3} it can be defined as a predicate that is satisfied on a state where *d3* is open. We have to check that such a state is reachable. This is done by formulating a sequential goal structure as shown below. It specifies the shortest scenario that could make *d3* open:

SEQ(*closeBy*(*b1*), *interacted*(*b1*),
closeBy(*d1*), *closeBy*(*b3*), *interacted*(*b3*),
closeBy(*d2*), *closeBy*(*b4*), *interacted*(*b4*),

[3] https://github.com/iv4xr-project/labrecruits

closeBy(*d3*),
assertTrue(..., check that d3 is open)

If this goal structure can be completed, the test is passed. Else it fails.

Arguably the test formulation above requires the full sequence of high level transitions that solves φ_{d3}. One might wonder if it is possible to generate this sequence, rather than having the developer manually specifying it. This will be discussed in the next subsection. However, do note that even if the sequence has to be given, they refer to high level transitions. The translation of such a transition to actual runs of deliberation cycles, and how the agent should choose which primitive action to execute (which is not trivial!) at each cycle, are abstracted away from the testers' concern.

Tactics and goal structures are very expressive, but we do need to program a set of basic goals (and their tactics). These are quite domain specific; so tactics that work for one SUT may not work for another (though their design patterns might be common). However, programming tactics is a one off investment. Once provided, we can keep using them to automate various testing tasks.

1.4 ONLINE GOAL SOLVING WITH CONSTRAINED OBSERVABILITY

In Section 1.3 we have observed that basic goals can be treated as **high level transitions**. The corollary of this is that solving a goal φ can be seen as a problem of finding a sequence of high level transitions that leads to a state satisfying that goal. In Section 1.3 we require the developer/tester to provide the solution. We will now discuss high level solvers that can search for such a sequence of high level transitions, so that the developer no longer needs to manually provide it.

We can notice that the basic tactics given as examples in Section 1.3 are parameterized by a target object; essentially they specify what the agent can do with the object, e.g. to travel to it, or to interact with it. The state of iv4xr agent contains information about objects it most recently observes. If the observation is unlimited (which is usually not the case), the agent can observe all objects in the SUT's virtual world. Else, only some limited objects can be observed, given the agent's current location. When the observability is constrained, the agent's state also stores information about objects it observed in the past (the mechanism is explained in more details in Report **D2.4**). Given knowledge about what objects are known to it, the agent would then also know which high level transitions are available at its current state. So it then can autonomously try different high level transitions in order to try to solve a goal.

Just randomly trying different transitions is of course not very productive. Also keep in mind that online execution is much more expensive than offline execution. A more systematic algorithm is shown in **Figure 1.5**.

The algorithm SA1 [Shi+21] takes a goal of the form φ_f where *f* is an object. It is a predicate that checks the state of the object *f*. E.g., if *f* is a door, φ might be checking if the door is open. It

iterates over objects *o* currently available in the agent's knowledge and reachable from the agent's current state. It would then travel to such an *o* and interact with it, and then check the state of *f* again if it now solves the goal φ. Each candidate will only be tried once, and a heuristic can be given, e.g., to try a candidate that is closest to the agent first, or closest to *f* first. The algorithm also incorporates exploration if no untried candidate can be found in the current state.

Algorithm *SA1.solve*(φ_f):
visited = ∅
while *true*
  if runs out budget **then return** fail // the goal is failed
  do *closeBy*(*f*)
  S ← the agent current state
  if φ_f(S) **then return** success // the goal is solved
  repeat
    U = {*e* | *e* is an object known in *S*} / *visited*
    if *U* == ∅ **then**
      if there is still unexplored and reachable terrain **then** do *explore*()
      else return fail
  until *U* ≠ ∅
  take an *e* from *U*, e.g., the closest to the agent
  do **SEQ**(*closeBy*(*e*), *interacted*(*e*))
  visited = *visited* + {*e*}

Figure 1.5: t*he online search algorithm SA1. The algorithm takes a goal of the form* φ_f *where f is an object. It is a predicate that checks the state of the object f. E.g., if f is a door,* φ *might be checking if the door is open. The algorithm is formulated imperatively in the usual algorithmic-style. In the implementation it produces a goal-structure and hence can be combined with other goal-structures.*

SA1 can only solve a goal that can be solved with a *single* high level transition; the algorithm essentially tries to find this solving transition. So, it will not be able to fully solve the door *d3* testing task from Section 1.3, but it can solve its fragments. With SA1 the testing task can now be formulated as follows:

SEQ(*SA1.solver*(*d1 is open*),
  SA1.solver(*d2 is open*), *closeBy*(*b4*), *interacted*(*b4*),
  closeBy(*d3*),
  assertTrue(..., check that d3 is open)

Notice that now it is shorter than the original formulation in Section 1.3.

A more advanced version of SA1, let's call it SA2, has been developed and studied [Shi+22]. It will be able to solve certain goals whose solutions require a sequence (of high level transitions)

of length >1. In **Figure 1.6** some results on the performance of SA2 on various levels[4] in the Lab Recruits game. The first eight levels are taken from the ATEST 2021 Testing Competition benchmark[5]. These are small to medium sized levels of 150 - 600 m^2. The last two are levels simulating two actual levels from an MMORPG game called Dungeon and Dragons online (DDO)[6]. The original DDO levels are called *Durk's Got a Secret* and *Return to the Sanctuary*, which we simulate in Lab Recruits. **Figure 1.7** shows the original layout of the DDO Durk level, alongside its simulated layout in Lab Recruits. These are large levels (Durk is 1600 m^2 and Sanctuary is 3400 m^2).

For each of these levels a non-trivial goal is chosen and given to the SA2. It manages to solve all of them. The time needed to solve is given in the table in Figure 1.6. DDO levels took more time to solve, because they are bigger. The column "tried doors" indicated the number of intermediate doors that the agent tried in order to solve the given SA2-goal (the main goal). Note that at the beginning the agent does not know how many doors there are in the level. It starts with almost zero knowledge, with just some approximate location of where the goal could be, as knowledge.

level	tried doors	time (s)	exploration (s)
LR3_1_1_H	3	68	10(0.14)
R4_1_1	5	84	19(0.22)
R4_1_1_M	5	139	24(0.17)
R5_2_2_M	6	140	27(0.19)
R7_2_2	4	146	42(0.28)
R4_2_2	1	60	20(0.33)
R4_2_2_M	4	144	41(0.28)
R7_3_3	6	254	58(0.22)
durk	4	2680(1800*)	1085(0.40)
sanctuary	16	1492	358(0.25)

Figure 1.6: *Results of goal solving with SA2 on various Lab Recruits levels. On all these levels SA2 manages to solve the chosen goal. "Time" is the time needed to solve the goal. "Exploration" is the time spent on exploring the level. As in SA1, exploration is a key part of the algorithm when the current state offers no further candidate to try.*

[4] In gaming jargon, a "*level*" refers to an instance of the same game, but played in a different world. The game mechanics stay the same, but a level would have its own unique world layout, objects, and logic between these objects.

[5] https://a-test.org/a-test-2021/

[6] https://www.ddo.com/home

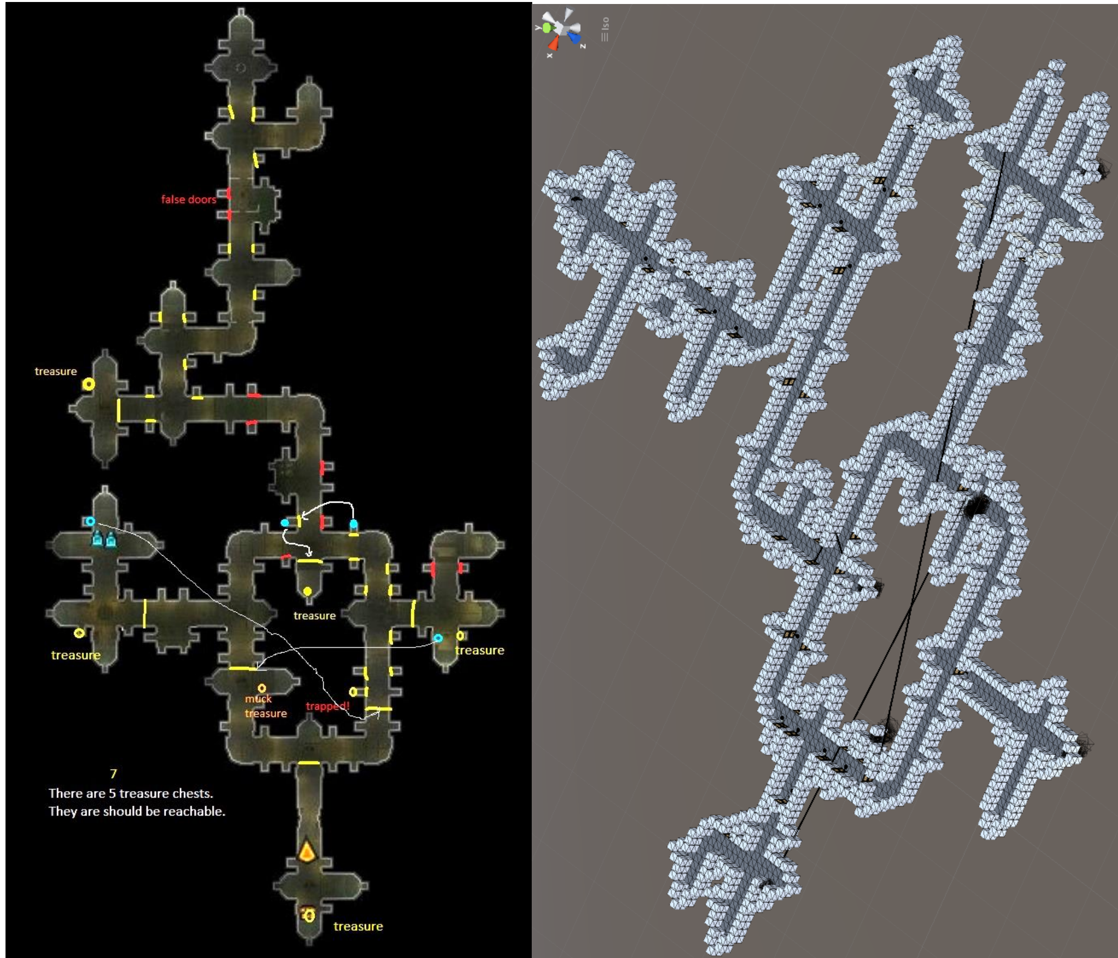


Figure 1.7: *on the left is the layout of the actual DDO Durk level. The picture has been annotated: yellow stripes/lines are doors (closed at the beginning). There are levers next to most of these doors that would open the door close to it; they are not marked in the picture. Blue circles mark key levers that would open important doors. Red are fake doors that cannot be opened. To the right we see the layout of the simulation of Durk in Lab Recruits. Fake doors are bad for SA2. If it tries to open one, because it speculates that there might be something useful it can use behind that door, this would trigger an inner loop of trying out various switches only to conclude at the end that it is better to forget the door and try another one.*

It is actually quite surprising that all these levels are solvable by SA2 (by this we mean goals similar to opening a door are solvable, for any door in these levels). There are conditions tied to SA2-solvability. We expect that most levels intended to be playable by a single human player should be SA2-solvable, but not all. For example, the goal to open *d3* in the testing task from Section 1.3 is actually not SA2-solvable. Some of the conditions that determine SA2 solvability are also difficult to understand, as they are delicately related to the geometry of the level/world and the agent's visibility constraint. SA2 is currently still under further study, to gain better understanding of its solvability conditions and to also improve its performance (e.g. employing multiple agents might greatly improve its performance).

A working prototype of SA2 exists along with experiment results. Its integration into the iv4xr Framework is postponed to wait for further results, but it is planned for the first quarter of 2023.

2. Reinforcement Learning FTA

2.1. Introduction

Reinforcement Learning (RL) is one of the main machine learning paradigms alongside supervised learning and unsupervised learning. RL focuses on learning the best course of actions (adaptive strategy) on an intelligent agent by exploiting its interaction data within a dynamic environment. Contrary to supervised learning which relies on labeled datasets, the RL training operates on the agent-environment loop (see **Figure 2.1**) where the action of the agent can alter the state of the environment. RL is researched in many domains such as games (board games, video games), robotics, telecommunications, etc.

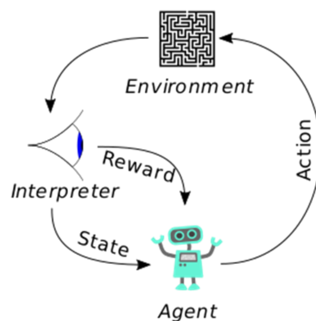


Figure 2.1: The reinforcement learning interaction loop

Reinforcement learning can therefore be used to interact with an XR system, learn gradually from this interaction, and ultimately be able to solve tasks related to that XR system. Here, we are especially interested in testing the functions of XR systems. RL based approaches have the potential to significantly improve automated testing as they have the capability of learning directly by interacting with the dynamic and uncertain XR system environments without the explicit need of modeling it.

We investigate two ways to implement RL for Functional testing:

1. Reinforcement Learning based Testing (RLbT) - RL solutions for automated testing and providing functional coverage of XR systems.
2. The RL agent is an opponent to the system and tries to adaptively find weaknesses in the system under test. The Quality-Diversity Reinforcement Learning (QDRL) techniques described below find multiple and diverse weaknesses at once.

2.2. REINFORCEMENT LEARNING BASED TESTING (RLBT)

This section describes the use of RL solutions for automated testing and providing functional coverage of XR systems. The use of RL solutions in automated testing of complex XR systems is challenging as it involves a vast amount of critical thinking, problem-solving, and path planning. In particular, in XR systems, the imperfect information of the agent due to its partial visibility of the environment, the large state-action space due to the long time span of the system, and delayed or sparse reward assignment pose challenges to effective RL solutions. To address these issues, careful attention is needed to represent states and actions and to define effective reward mechanisms in the environment.

To this end, we have used a curiosity driven reinforcement learning approach where we remain at a higher level of abstraction when defining the states and actions of the reinforcement learning environment and with a curiosity-based reward scheme that has the ability to become a powerful exploration mechanism to facilitate the discovery of solutions for complex, sparse or long-time span tasks. Specifically, the scheme is beneficial in this context where we aim to maximize the functional coverage. This tactic of using curiosity is more generic as it can be applied to diverse environments. This reward mechanism enables the reinforcement learning agent to explore the space of interactions in the game. It encourages the discovery of previously unseen states and discourages immobility and revisiting of already seen states.

Empirical evaluation is carried out by applying RLbT on a 3D maze-puzzle game called Lab Recruits. Details about the experiments are presented in [FRK+22]. To assess the feasibility and effectiveness of reinforcement learning solutions in automated game testing and coverage, we compare the proposed curiosity-based RL solution with two alternative baseline solutions. First is sparse-reward RL, a classic RL approach with only intrinsic sparse reward, where an agent receives positive feedback only when it reaches its goal, otherwise nothing. The second baseline approach is the pure random solution, where the agent takes decisions randomly.

We carried out our experiments on five levels of Lab Recruits with different characteristics that allow us to get insight into the applicability of RL in automated testing. **Figure 2.2** shows two among those levels, **Figure 2.2(a)** presents a level containing 32 buttons and 16 doors distributed across 8 rooms. The layout of the level is in such a way that the agent can fairly easily spot the buttons and eventually observe their effects (i.e., doors that open/close) as well. **Figure 2.2(b)** presents a randomly generated level where the number of entities is comparable to 8-room (19 buttons and 14 doors distributed across 14 rooms connected by long corridors) but the physical space covered by the level is significantly larger. This means that the agent needs to travel a long distance to get from one entity to another. This level poses a different type of difficulty to reinforcement learning also because observing the effect of an action (e.g., a button pressed) is difficult as the corresponding door that is opened/closed may not be immediately visible as it resides in a different room across a long maze of corridors.

Figure 2.2: (a) 8-room level **(b)** LargeMaze_1 level of Lab Recruits

Several parameters control different aspects of RLbT. Some of the parameters are related to reinforcement learning, and some are related to the SUT and the test agent used to interact with it. Based on our preliminary experiments, we selected parameter values that represent a reasonable tradeoff between effectiveness and efficiency without much loss in the generalizability of RLbT.

In our experiments, we first discuss the results related to the goal oriented exploration followed by the coverage oriented exploration. With goal oriented exploration, the aim is to test the game-play, that is to learn the best way to achieve the specified goal in the game. For the levels of Lab Recruits we used in our study, this translates to activating a sequence of buttons that open various doors until the specified target goal is reached, in our case a specific door is opened. Our experiments on various levels of Lab Recruits shows that the agent is able to effectively learn the optimal sequence of actions needed to achieve the goal.

In measuring the functional coverage achieved by the RL agent, we concentrate on identifying the following coverage metrics important for the Lab Recruits game and the approach to measure the quantitative value of coverage achieved by our explorative agent.

- Entity coverage - percentage of observed/interacted entities (with all possible properties) in a level of Lab Recruits. For example, the level of Lab Recruits (as shown in **Figure 1.4**) features fourteen properties of seven entities (i.e., three doors and four buttons). A door can be observed in two statuses, thus having two properties Open and Closed. A button can be observed in two statuses, thus having two properties: pressed or not-pressed.
- Entity Connection Coverage- In a level of Lab Recruits, doors are usually connected with buttons. This metric measures the ratio of connection satisfies in a level. Measuring the quantitative value of Entity Connection Coverage is difficult due to the partial observability issue of the agent. We follow a probabilistic approach to measure this metric.

Figure 2.3(a) and **2.3(b)** show the per episode entity coverage achieved during the training phase for 8-room and LargeMaze_1 level. It is noticed that curiosity-based RL shows good coverage results in a small and straightforward level like 8-room, while it shows significant improvement in achieving entity coverage compared to sparse-reward RL and random solution in LargeMaze_1 level.

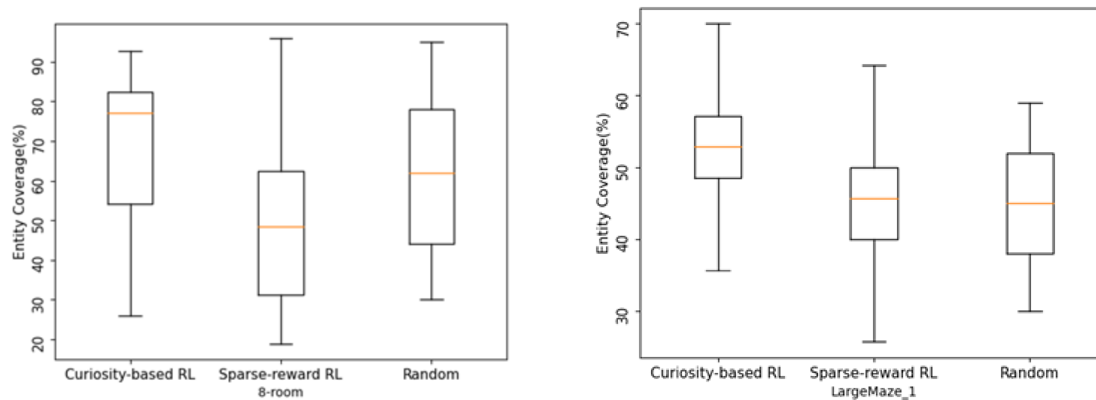


Figure 2.3: Entity Coverage per episode **(a)** 8-room level **(b)** LargeMaze_1 level of Lab Recruits

We have run a test game-play session to measure the quantitative value of coverage. The result is compared only between two RL solutions as the test session is guided by the respective learned policy. Our goal is to observe the coverage ratio obtained by both RL solutions with a limited budget. Coverage results obtained from our experiments are presented in **Table 2.1**. It is noticed that for a simple and small level like buttonDoors both curiosity-based and sparse-reward RL achieve high/full coverage. While curiosity-based has shown high potential to obtain better coverage ratio in a large and complex level like LargeMaze_1. Though both RL solutions obtain low coverage for LargeMaze_2 level, this may be because the learning duration was not enough to acquire an optimal Q-table.

Environment Level	Curiosity RL		Sparse Reward RL	
	Entity Cov	Connection Cov	Entity Cov	Connection Cov
buttonDoors	100%	100%	100%	100%
4-room	100%	80%	60%	50%
8-room	92%	60%	80%	54%
LargeMaze_1	80%	50%	40%	37%
LargeMaze_2	55%	30%	40%	20%

Table 2.1: Coverage measure of difference levels of Lab Recruits game

RLbT can be run in multi-agent mode where it deploys the Multi-Agent Reinforcement Learning (MARL) architecture. MARL architecture consists of a group of autonomous, interacting agents sharing a common XR environment to achieve common or conflicting goals.

In MARL, the agents are autonomous entities with individual goals and independent decision-making capabilities, but they are influenced by each other's decisions as they apply reinforcement

learning in a shared environment. This makes the learning of an optimal policy inherently difficult in MARL. Despite the added learning complexity, a real need for multi-agent systems exists. Particularly, for complex, physically large and inherently decentralized XR systems where a single agent learning approach is not effective.

In multi-agent mode, RLbT deploys two agents that work in a collaborative manner in a shared XR environment with the goal of maximizing coverage. The agents follow two behavior profiles: active and passive. An *active agent* is guided by a curiosity-driven reward mechanism to explore the system and produce a set of actions that cover the various elements in the system as well as functional aspects such as the connection between doors and buttons, for example. To help the active agent, RLbT deploys another *passive agent* that is responsible for scouting the environment and reporting its observations to the active agent. This enables the active agent to be aware of the effects of its actions in an efficient way, especially in systems where the environment is large and complex. For instance, in Lab Recruits where a button in one room could open a door in another room, the fact that there is a second agent, possibly far from the active agent, allows us to observe changes to the environment triggered by the actions of the active agent. The agents communicate and propagate information among themselves through sharing their observations.

The RLbT multi-agent approach is currently applied on Lab Recruits, exploiting its multiplayer feature. However, it could be applied to similar systems that support multiple players. We carried out our experiments on two levels of Lab Recruits: (a) a large level as shown in **Figure 2.2(b)**, and (b) an extreme level where the entities (buttons and doors) are around 5 times higher in quantity than the large level and are distributed over a large physical space. The performance of the multi-agent architecture is compared with that of the single agent. **Figure 2.4** and **Figure 2.5** shows the global coverage and the per episode coverage achieved for large and extreme levels. It is noticed that use of multi-agent features increases the global coverage for both levels, but it becomes beneficial as the level complexity increases.

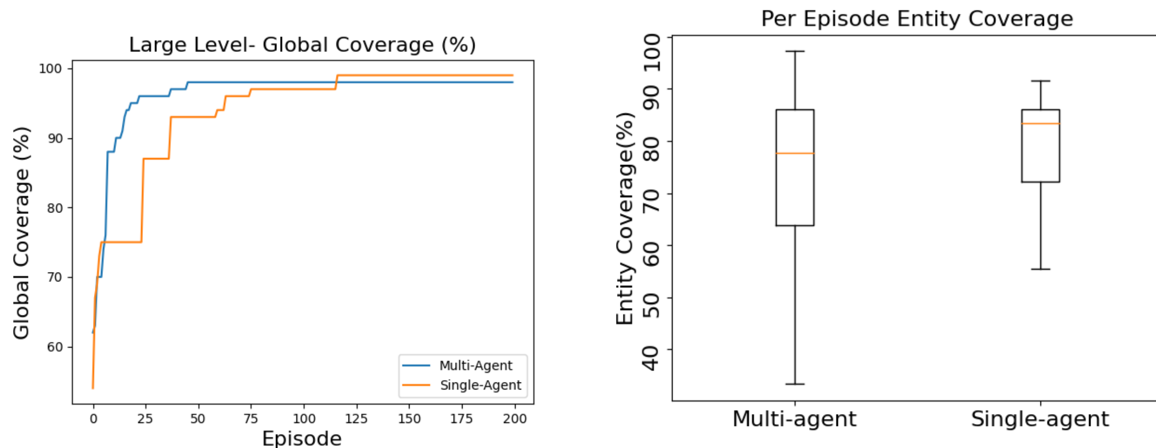


Figure 2.4: Performance comparison for Multi-agent vs single agent RL on LargeMaze_1 level of Lab Recruits **(a)** Global coverage **(b)** Entity coverage per episode

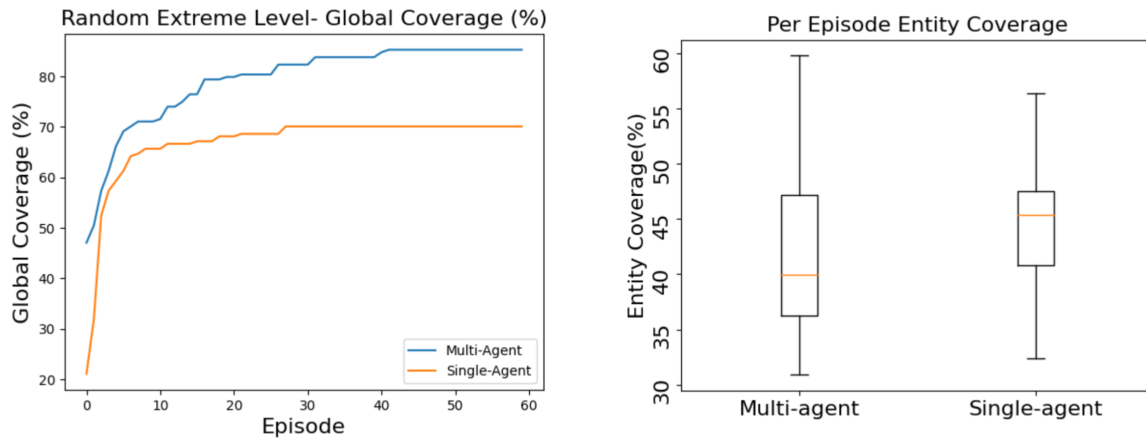


Figure 2.5: Performance comparison for Multi-agent vs single agent RL on Extreme level of Lab Recruits **(a)** Global coverage **(b)** Entity coverage per episode

2.3. Quality-Diversity RL (Thales MAEV)

This section tells how we use RL in order to find intrusion scenarios in a powerplant guarded by a moving patrol as illustrated in **Figure 2.6**. The agent needs to infiltrate the powerplant without being detected by the guards or the fixed camera. In this scenario we want to find multiple intruding strategies in order to find all the weaknesses of the guarding patrol.

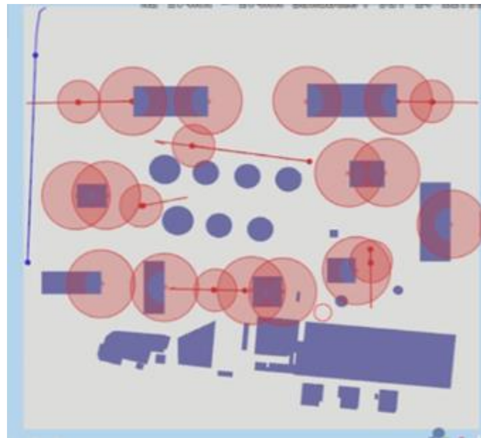


Figure 2.6: This image shows a simulation of the powerplant environment where the buildings are in blue, the camera and moving guards in red and the intruder is the blue trace in the top left (along with its path plan).

QD-RL is bringing the QD approaches to RL. Quality-Diversity is an approach to solving a problem that maintains a population of candidate solutions and that does not uniquely aim at improving their quality but instead will balance between looking for improved performance (quality) and increasing the diversity of the population of solutions (diversity).

This approach can help explore more diverse types of solutions and avoid that the learning process focuses too much on a type of solution that will not ultimately be optimal (avoiding falling in local minima). Also if the final requirement is to output all the solutions of the problem (or as much as possible), this approach is likely to output a lot of them.

In order to achieve this, QD-RL maintains a population of candidate solutions, called an archive, and enforces during the learning process that some of the solutions grow apart to favor originality. Therefore, designing and quantifying originality is an important component of the process.

To quantify originality one can distinguish two main approaches. The first one parametrizes the space of policies, partitions it, and registers the best policies for each cell of the partition. The second one defines the originality of a solution in a more relative way, relative to the other solutions in the population by computing a distance between policies. In our implementation we followed the latter as defined in [TP+22].

QDRL is a meta algorithm. It is based on a base RL algorithm, in our case TD3, that is able to solve the base problem but can only output one solution. QDRL works through iteration, maintains a population of strategies and creates new ones at each iteration of the algorithm. To create new strategies, QDRL selects among the old ones several that are on the pareto frontier of quality and diversity criteria. Half of them are improved with respect to performance criterion (using TD3) and the other half is improved according to diversity criterion (using also TD3 as the diversity is formulated as a reward and falls directly in the RL framework).

We implemented QDRL on a series of maze problems, whose complexity increases, leading up to the powerplant problem. Playing with simpler mazes allows us to tune our algorithm and discover its limits.

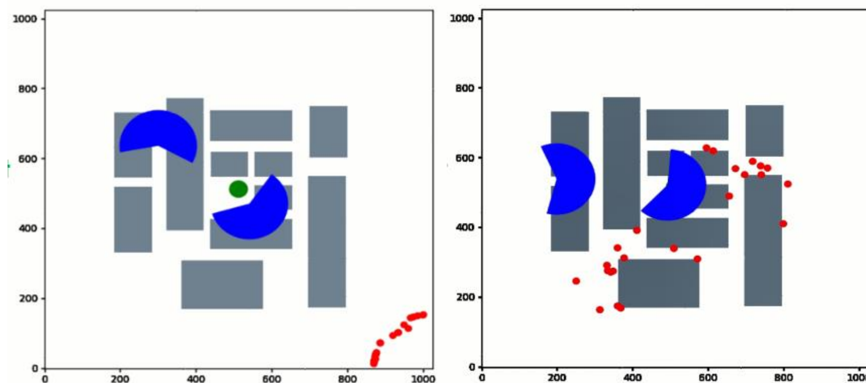


Figure 2.7: This shows the execution of several RL agents that successfully introduce the guarded maze with two guards using diverse strategies from the same starting point. Left: The RL agents start from the same corner and aim at the center of the maze (green circle). Right: The RL agents are getting closer to the target using different strategies to defeat the guards.

We evaluate the run of QDRL by looking at different metrics: how does the performance of the strategies in the archive evolve? How does the diversity of the archive evolve? In order to quantify

both diversity and quality at the same time we also look at the QD score which is the sum of performance of all the population in the archive.

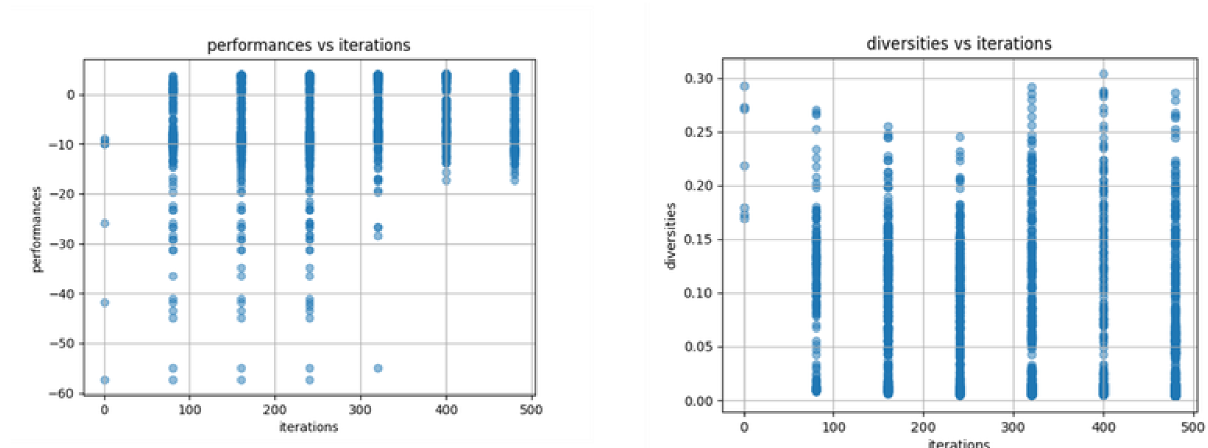


Figure 2.8: The evolution of the performance of the RL agents in the archive (left) and the evolution of the diversity in the archive (right) versus the iterations of the QDRL algorithm.

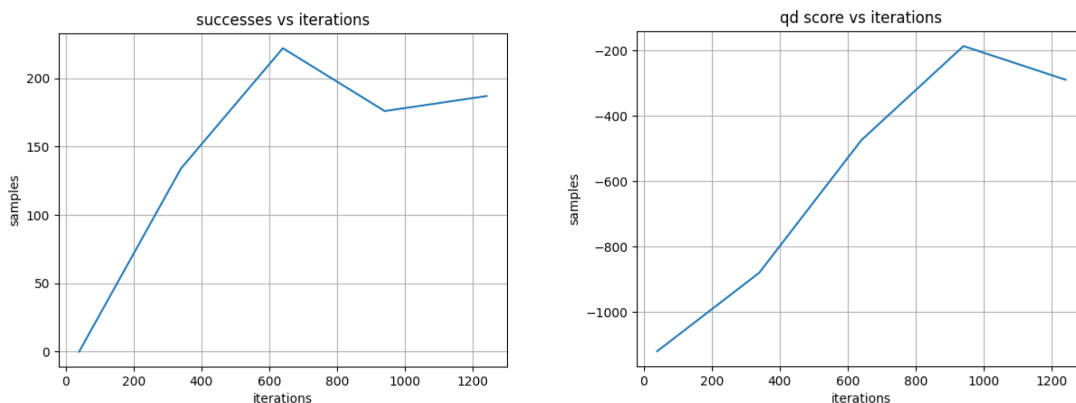


Figure 2.9: The evolution of the number of successful RL agents in the archive (left) and the evolution of the qd-score in the archive (right) versus the iterations of the QDRL algorithm.

Figure 2.8 (left) shows that the performance of the archive grows through iterations. This is consistent with the growth of the number of successful RL agents in **Figure 2.9** (left) and the growth of the QD-score in **Figure 2.9** (right). In **Figure 2.8** (right) we see that diversity starts low but very fast grows to a high level that is maintained throughout the experience.

Finally, to quantify the diversity of our agents we can display how these agents acquire an important coverage of the fields of intrusions.

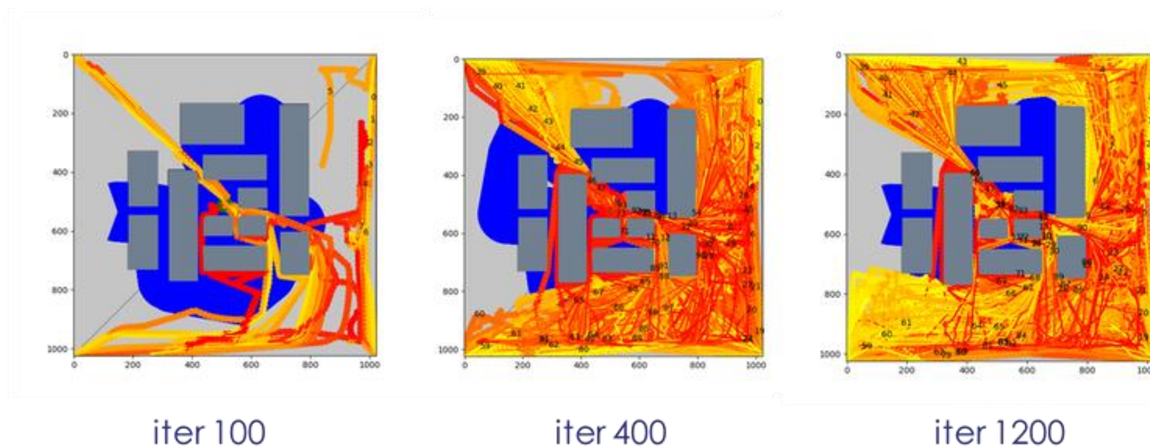


Figure 2.10: The evolution of the coverage of successful RL agents in the archive over the field of the maze versus the iterations of the QDRL algorithm.

We see in **Figure 2.10** that the QDRL algorithm is able to slowly increase its coverage until it achieves almost full coverage of all possible paths.

Finally, we tested QDRL in the complete scenario that is the power plant scenario illustrated in **Figure 2.6**. Our implementation of the QDRL algorithm fails to find successful RL agents in the power plant scenario. As explained above QDRL is a meta algorithm that trains a population of agents with the use of the RL algorithm TD3. We experimented with TD3 alone in the powerplant scenario, optimizing only for performance and being able to return only one solution. After one week of training, TD3 is able to output an RL agent with 90% intrusion success.

Replicating the QDRL training process over an archive of at least 20 agents requires a long learning time. For this reason, future work requires us to implement a parallel version of the QDRL algorithm spread over multiple processors.

3. Scriptless Exploratory FTA

3.1. Introduction

Exploratory FTAs do not follow specific instructions such as a set of tactics and goals or crafted models to interact with the XR System Under Test (SUT) and to test specific paths of XR interactions. The objective is based on the execution of non-sequential actions to test that the SUT and its functional aspects are robust enough to respond to different user interactions.

The automated testing approach of exploratory FTAs in the iv4XR framework is based on integrating the open-source **TESTAR** tool. This tool existed before the creation of the iv4XR project and emerged as a result of the European project FITTEST[7]. The underlying principle of TESTAR follows the scriptless approach to test Graphical User Interface (GUI) systems: generate test

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

sequences of (state-actions)-pairs by connecting to the SUT to obtain the State and continuously executing exploratory actions to bring the SUT to another State while applying Oracles to detect failures (see **Figure 3.1**).

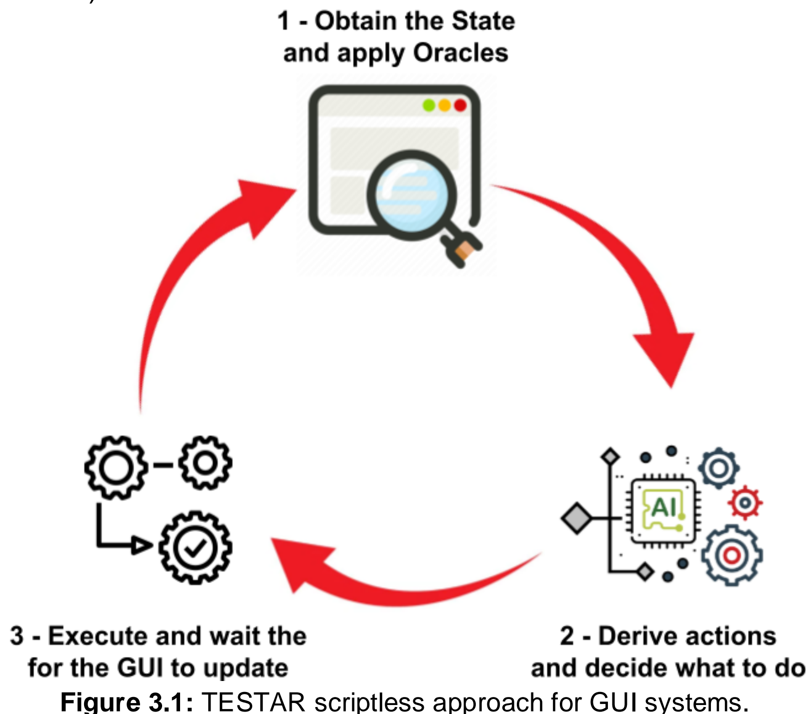


Figure 3.1: TESTAR scriptless approach for GUI systems.

An overview paper that describes the TESTAR capabilities to test GUI systems was published in the STVR journal in 2021 [TV+21]. Although the integration and use of TESTAR for XR systems were not described, we did acknowledge the iv4XR project in the paper (together with many other projects and people). That is because several revisions of the paper have been made during the iv4XR project execution. In the next sections, we describe how the TESTAR tool has been extended to act as an exploratory FTA within the iv4XR project.

3.2. TESTAR IV4XR EXTENSION

The TESTAR tool is modular software that allows the integration of multiple software frameworks, plugins or APIs extensions to adapt the concepts of State, Actions, and Oracles to test different types of systems by using the scriptless approach. While the core package of TESTAR is fitted with the concept of exploratory FTA, a series of technical extensions were incorporated into the tool to integrate it as an agent within the iv4XR framework.

1. TESTAR uses the concept of **`Tags`** to set and get generic variables `<T>` to the `Taggable` classes that represent the SUT, State, Widget, and Action objects. These `Tags` that consist of pairs of `<name, value>` properties are defined in specific `API-Tags` classes (e.g., `UIATags`, `WebTags`, `AndroidTags`, etc.) and customize specific systems properties (e.g., `WebTagName`, `WebCssClasses`, `UIAControlType`, `AndroidResourceId`, etc.) that are used to indicate TESTAR which actions derive or which oracles apply.

To extend TESTAR within the iv4XR framework, a new `IV4XRtags` class was added to the tool. To fetch the state of XR systems iteratively and to execute interactions, `Environment`, and `Controller` Tags were added to the classes representing the SUT objects. To allow TESTAR to map the entities' properties that come from XR systems in order to derive actions and apply oracles, multiple Tags that represent the `Position`, `Orientation`, `Size`, or `Integrity` of the XR entities or the `Health` and `Energy` of the XR agent were added to the classes that represent the State and Widget objects.

In **Example T1**, TESTAR uses the SE-plugin controller presented in ***Deliverable 5.4*** with the name `iv4xrSpaceEngineers` in the SUT object class to obtain the Character interface that allows FTAs to execute actions such as using Space Engineers (SE) tools. Then, in **Example T2**, TESTAR uses the `seFunctional` property to determine whether a navigate and interact action must be derived for each widget block observed in the SE state.

```
SpaceEngineers seController = SUT.get(IV4XRtags.iv4xrSpaceEngineers);
Character seCharacter = seController.getCharacter();
seCharacter.beginUsingTool();
```

Example T1: Usage of Tags to obtain the SE controller and execute a SE tool action.

```
for(Widget widget : state) {
      if(widget.get(IV4XRtags.seFunctional)) {
            seActionNavigateInteract(w);
      }
}
```

Example T2: Usage of Tags to derive navigate and interact actions for functional SE blocks.

2. The **`State`** of TESTAR, which represents which virtual entities are observable in a specific range, consists of a hierarchical set of widgets with properties known as widget-tree. To obtain the state of XR systems, TESTAR realizes an observation of the SUT using the iv4XR framework, which allows the tool to obtain the World Object Model (WOM) information (see **Figure 3.2**).

Figure 3.2: TESTAR observed State from Space Engineers system.

This implementation to fetch the WOM information through the iv4XR framework and create the state is implemented by creating a new iv4XR module which extends the core module of TESTAR (see **Figure 3.3**). `LabRecruitsProcess` and `SpaceEngineersProcess` are the new classes that contain the functionality to launch the XR SUT and desired scenario, together with the initialization of the XR environment and controller, which allows TESTAR to communicate with the XR system. When the SUT starts, and after each action execution, TESTAR uses the classes `LabStateFetcher` or `SeStateFetcher` to use the attached environment and controller `IV4XRtags` to realize the observations that allow the tool to obtain the WOM information and create the TESTAR State.

3. In order to interact with the virtual entities that are represented as widgets in the state, TESTAR needs to derive different types of **`Actions`** for the different types of interactive entities (see **Figure 3.4**). The new TESTAR iv4XR module contains two new main types of actions: basic action commands and compound action goals. A basic action command is the most basic event that TESTAR can execute using the iv4XR framework, e.g., move or rotate one step, equip or use a tool. However, due to the essence and complexity of XR systems, most of the time, it is necessary to execute a compound action goal that allows the exploratory FTA to navigate, aim and interact with an entity using an SUT tool.

Compound actions goal contains an additional `TestAgent` taggable object to allow TESTAR to execute tactics from the goal-solving agents as actions. This helps to reduce integration effort if the SUT is ready to run with this other type of FTA.

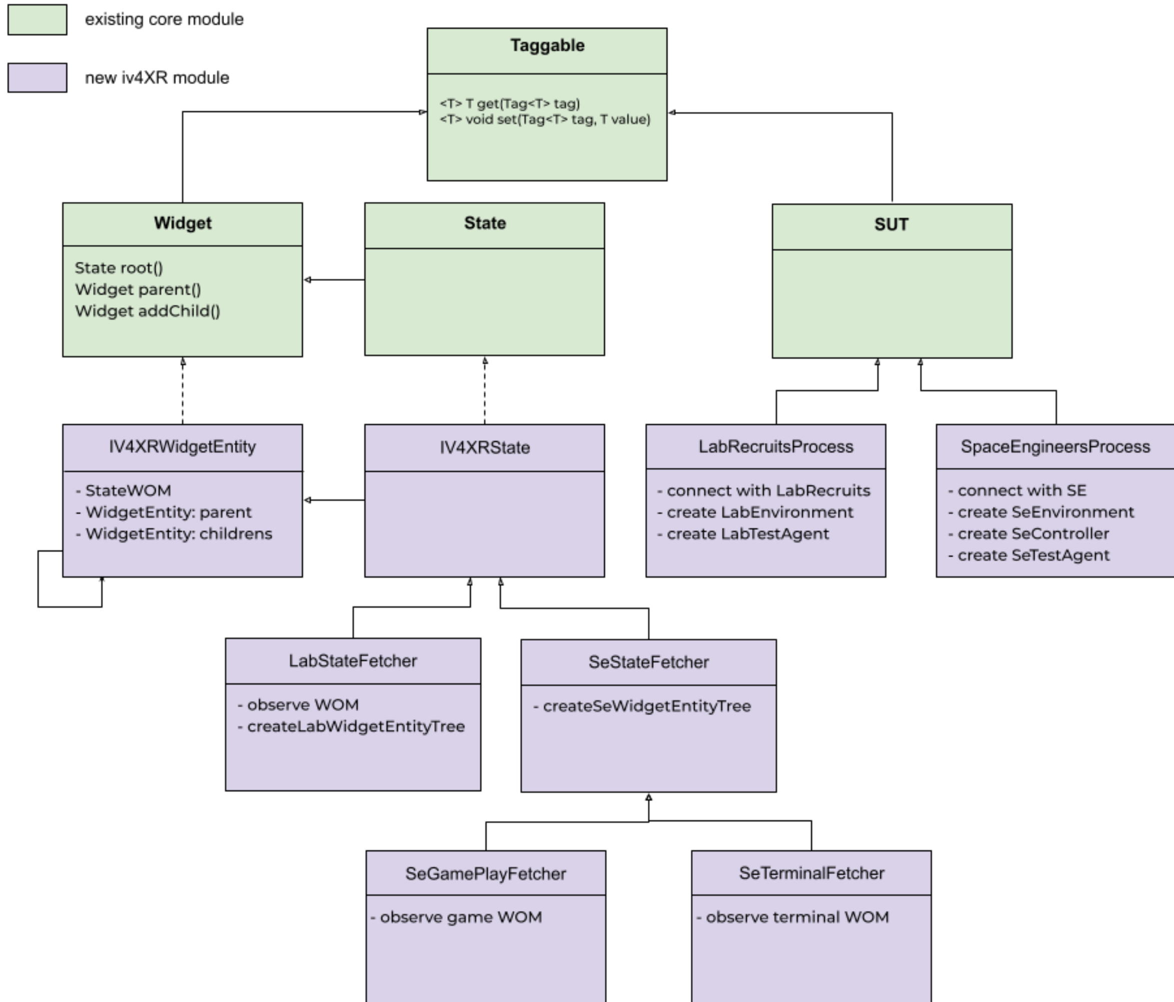


Figure 3.3: TESTAR iv4XR state module.

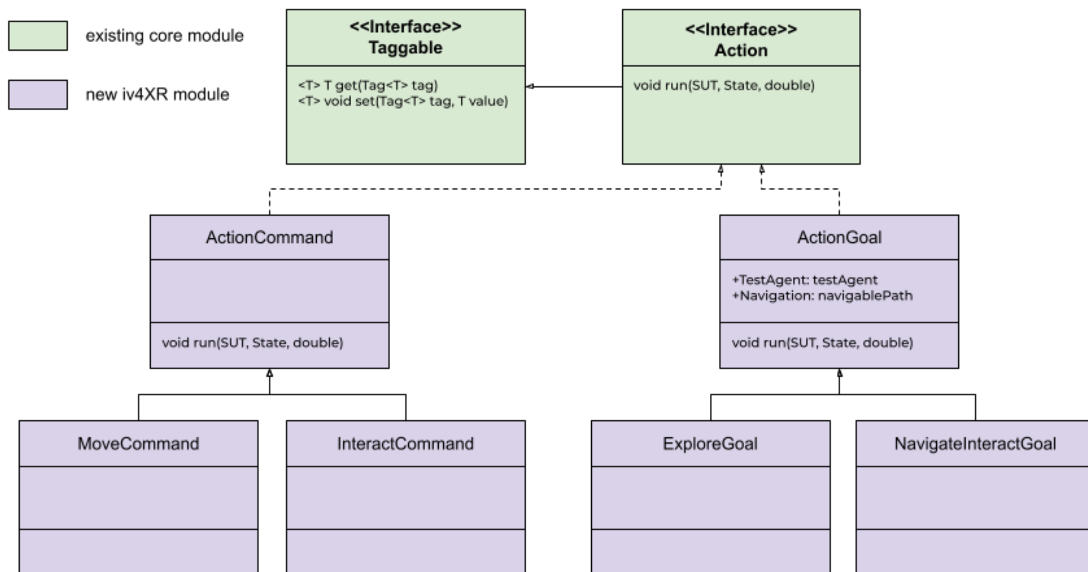


Figure 3.4: TESTAR iv4XR action module.

4. The core module of TESTAR integrates generic **`Oracles`** intended to verify the robustness of the SUT to detect if the system process has crashed or hung. Within the context of iv4XR, the TESTAR tool has been adapted to check for suspicious messages in the entities that are part of the game, as well as their terminals and panels, and to analyze for the exception messages in external SUT logs, as in the case of SE.

Although verifying the robustness of the SUT gives effective results when testing GUI systems, for XR systems, it is of great importance to test the functional aspects of the virtual entities. We extended the tool with new oracles to make the exploratory FTA capable of detecting functional failures for XR systems.

In the SE environment, it is essential to verify that the integrity of all type of block increase or decrease correctly after interacting with different tools, that the agent health, oxygen, hydrogen, and energy is restored when interacting with medical rooms or cockpits, that the jetpack and the dampeners are not switched without player activation, or that it is possible to construct new blocks if the player has the materials (see **Figure 3.5**).

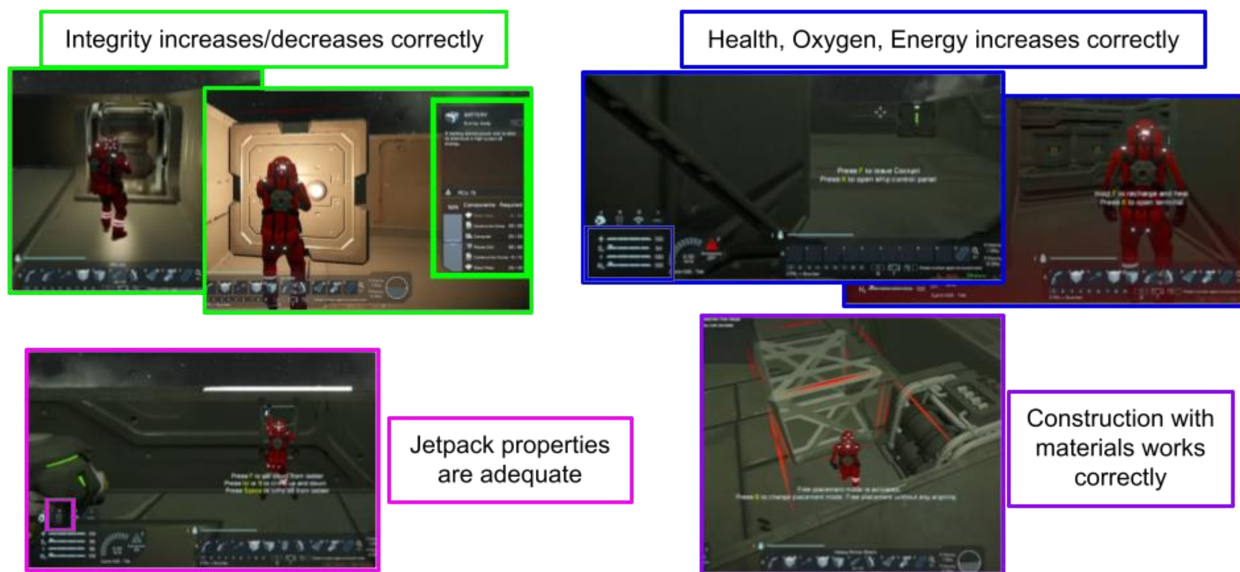


Figure 3.5: TESTAR oracles to test the functional robustness of the SE environment.

5. TESTAR can infer a **`State Model`** while in a state `s`, it selects and executes an action *a* and obtains a new state `s'`. The transitions *(s → a → s')* are then stored in the model until the tool stops the exploratory process. Extending the model to use the new `IV4XRtags`, `iv4XR State`, and `iv4XR Action` allows TESTAR to infer a state model while exploring XR systems.

This model can be beneficial for the stakeholders to visualize the model transitions, query the model and create offline oracles, or apply more intelligent action selection decisions as Reinforcement Learning (RL) strategies by remembering what states were discovered and which actions were executed. However, the TESTAR state model was originally designed for traditional GUI systems, on which the states contain all the available widgets to interact with.

In XR systems, the agents have the possibility of navigating around the virtual environment in order to reach the interactive entities. Because the entities that the agent observes are determined by an observation range, the position of the agent, and potential blocking elements, the exploratory FTA TESTAR needs to explore the navigable areas of the XR environments while storing the position of the entities to learn which of them are reachable (see **Figure 3.6**).

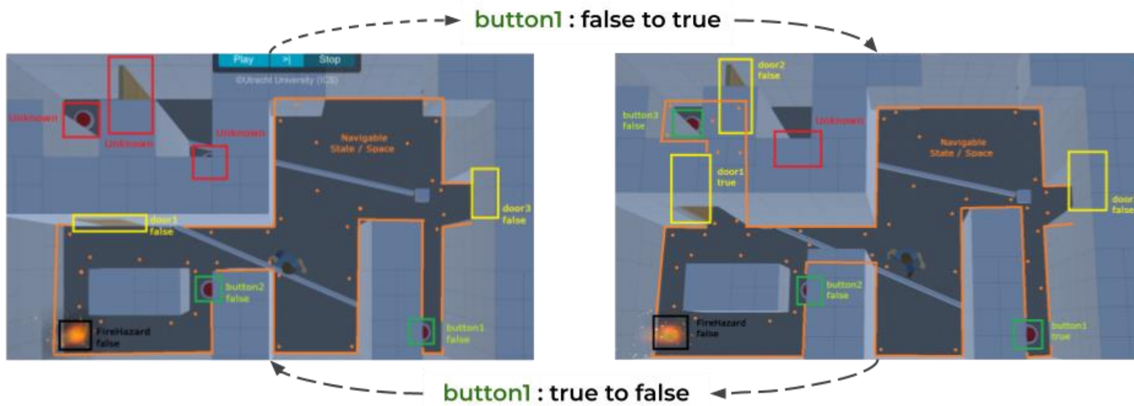


Figure 3.6: Navigable State/Space of the LabRecruits system.

For this reason, the TESTAR state model was extended with the concept of **`NavigableState`** (see **Figure 3.7**). 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 executes interactive actions defined as NavigableAction (e.g., open a door by interacting with a button), to continue with a new exploration of the available state positions.

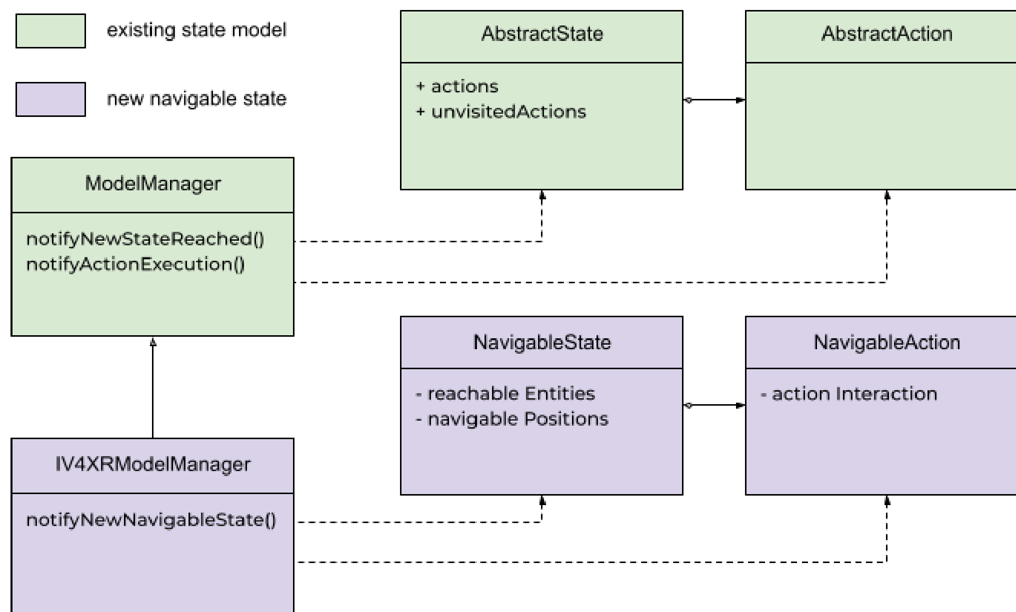


Figure 3.7: TESTAR iv4XR navigable state module.

6. The default **`Action Selection Algorithm`** of TESTAR makes stochastic decisions to explore the SUT. The extension of the `State Model` allows this FTA to remember which XR positions were explored and which XR entities interacted to prioritize the selection of `Unvisited`

`Actions` trying to expand the exploration coverage in terms of exploring different areas and interacting with different entities. Although this approach helps to interact with as many different entities as possible, there is no learning phase to consider other possible entity features, such as entities that provoke changes in the environment or in the agent itself (e.g., realizing an exploration that learns which interactions modifies the agent fuel to take a posterior decision to test the functional jetpack feature).

We have developed a new **`RL framework`** in TESTAR that allows attaching additional `RLTags` in the `State` and `Action` objects of the `State Model` to automatically calculate and assign reward values and learn from the previously executed transitions. This framework does not follow a specific RL strategy. Instead, it allows stakeholders to implement their `Policy`, `Reward`, and `QFunction` strategies to adapt the learning phase for different types of SUTs (see **Figure 3.8**).

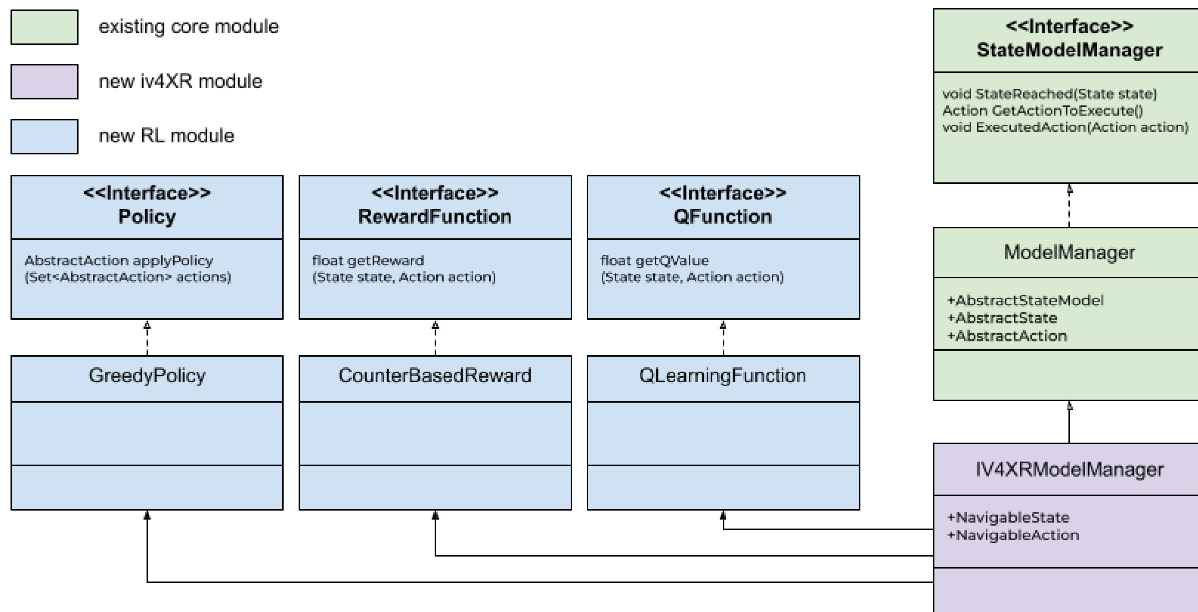


Figure 3.8: TESTAR new Reinforcement Learning framework.

7. The `State Model` inference benefits the exploration process of TESTAR by prioritizing the selection of `Unvisited Actions`. However, for large XR environments, inferring a model requires a long execution time. In order to speed up the exploratory process, we have developed a **`Distributed approach`**.

Multiple TESTAR instances can now connect to a centralized state model to share the knowledge of the observed environment. This is possible due to the usage of the same abstraction mechanism used in TESTAR to identify states and actions using the widget properties. A new Action Selection Mechanism (ASM) allows all TESTAR instances to coordinate their action selection by marking the target actions they pretend to execute.

Figure 3.9 represents how three different instances coordinate to select actions that were discovered but not yet executed. While Instance 1 can mark the closest red-action to execute, Instances 2 and 3 need to coordinate in the action selection because they are both in the same state. Instance 3 first marked the orange-action to execute, then Instance 2 read this mark information in the central model and decided to mark the blue-action that exists one state away.

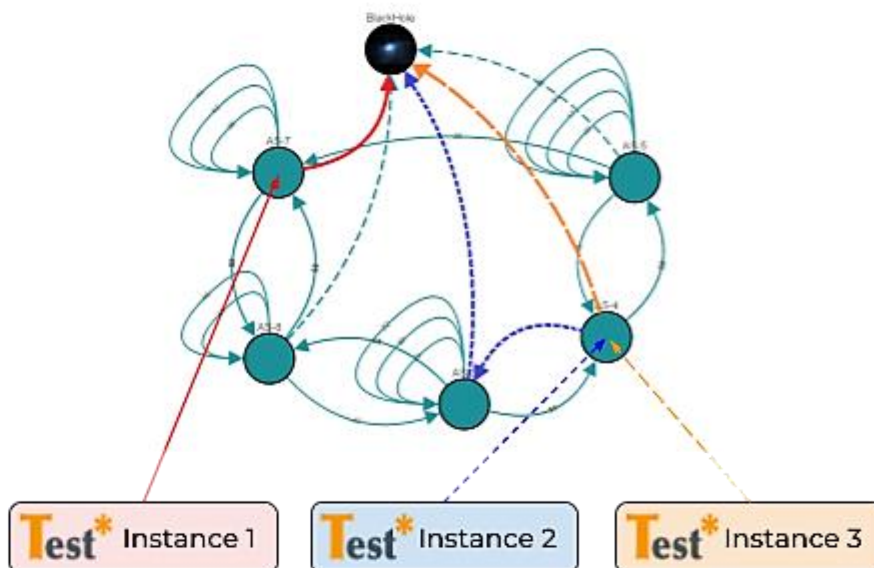


Figure 3.9: TESTAR distributed state model inference approach.

3.3. TESTAR OUTPUT FILES

As the exploratory agent TESTAR observes, navigates, and interacts with the XR entities, it can create four different types of output files (see **Figure 3.10**) that stakeholders can use to analyze the testing process and to obtain detailed information about potential detected failures.

1. **`HTML report:`** In each action iteration, TESTAR creates this action-by-action report with visual and textual information. It uses the Windows API to obtain a screenshot of each observed state and the information obtained through the iv4XR framework that indicates the existing interactive entities in those states.

2. **`State Model:`** As we have mentioned, TESTAR infers a State Model while exploring the XR system. The tool also offers an Analysis mode that launches a web service that allows visualizing the states and actions of the model.

3. **`Spatial Map:`** In some XR systems, such as LabRecruits or SE, the information of the level representing the environment on which the FTA will realize the exploration is stored in a local file that indicates the size and the existing elements. For these cases, it is possible to obtain Spatial Coverage metrics and create a visual map indicating which space was covered by TESTAR and which entities were observed and interacted with. ***Deliverable 5.4*** contains a use case example of spatial coverage metrics with SE.

4. **`Code Coverage:`** It is possible to integrate the usage of code coverage software in TESTAR to obtain lines and branch coverage metrics that indicate which internal code functions were executed during the TESTAR exploration. For example, for SE, TESTAR can automatically download and execute the OpenCover[8] software to obtain these metrics.

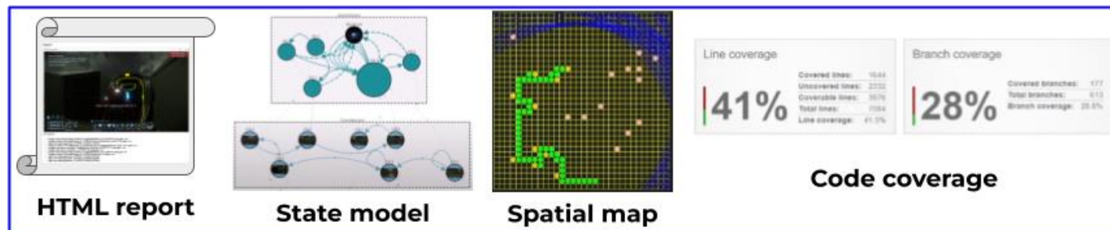


Figure 3.10: TESTAR output files created during the testing exploration.

3.4. References for documentation, videos and papers

The wiki section of the TESTAR_iv4xr GitHub repository contains technical details regarding the architecture of the TESTAR software and instructions with videos for the configuration and usage of the tool.

- https://github.com/iv4xr-project/TESTAR_iv4xr/wiki

The iv4xr-framework GitHub repository contains documentation indicating the scientific publications related to TESTAR and instructions about how the tool can be used as an exploratory FTA to test LabRecruits and Space Engineers systems.

- https://github.com/iv4xr-project/iv4xr-framework/blob/main/docs/testar/TESTAR.md

4. Coverage

The functional test agents implemented in the iv4XR project support different coverage criteria. Coverage informs testers how much a system is exercised by a test suite. The notions of functional coverage supported by the iv4XR test agents include:

- **Code coverage**. Code coverage represents the portion of the SUT's source code executed when a test suite is evaluated on the SUT. As all the iv4XR test agents support the execution on SUT, code coverage can always be collected.
- **Spatial (or area) coverage**. An XR system typically includes a representation of a 3D space. Spatial coverage refers to the fraction of the space the testing agent can explore

[8] https://github.com/OpenCover/opencover

during test suite execution. TESTAR supports spatial coverage for the Space Engineers game (see D5.4). The goal solving agent (see Section 1) includes the notion of area coverage (see D3.4).

- **Model coverage**. When a model of the system is defined, several notions of model coverage can be considered. Both TESTAR (see Section 3.2) and EvoMBT (see Section 4.1) include a notion of model. Instances of model coverage supported by the iv4XR test agents include:
  - *state coverage*: a state represents an entity in the system. To satisfy state coverage a test suite has to explore all the entities defined in the SUT. Both TESTAR and EvoMBT support state coverage;
  - *transition coverage*: a transition models an action performed in the system by the agent. The transition coverage criterion is satisfied when all the actions in the SUT are executed. EvoMBT supports transition coverage;
  - *k-transition coverage* extends transition coverage requiring that all the possible sequences of actions of length k are executed. EvoMBT supports k-transition coverage.
- **Quality-Diversity coverage**. Quality-Diversity coverage refers to the proportion of the SUT interaction space exerted by the execution of a test suite. Quality-Diversity coverage can be seen as an instance of model coverage, where only the set of available interactions is considered in the model. Satisfying Quality-Diversity coverage is particularly hard in XR systems, as the interaction space could be huge. Section 4.2 presents an algorithm to tackle the problem for the Space Engineers pilot.

4.1. EvoMBT: Evolutionary Model Based Testing

EvoMBT combines model-based testing (MBT) with search algorithms for the automated generation of test cases for systems with complex and fine grained interactions such as XR systems.

4.1.1 Model-based testing

Model-based testing (MBT) is a well established field in automated testing where formal representations of a system under test (SUT) are used to drive the generation of tests satisfying various coverage criteria. In particular, when the SUT presents a high level of complexity, applying MBT could be beneficial as it helps reduce complexity making test generation manageable. Furthermore, abstraction via models allows one to focus on desired aspects of the SUT, further reducing complexity. In situations such as XR systems, where the environment is highly interactive, MBT offers an advantage from the test generation perspective as it allows to model and interact with only a specific aspect/scenario of the system. Eventually, additional SUT behaviors could be modeled and tested in an iterative manner until the desired testing goal is reached.

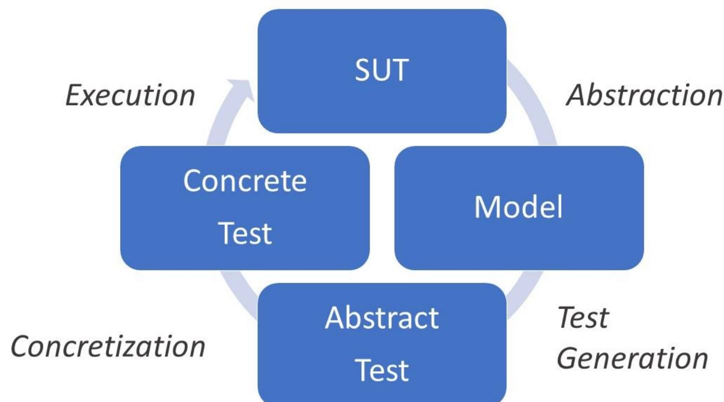


Figure 4.1: Model-based testing cycle.

MBT test generation cycle depicted in **Figure 4.1** comprises four main actions:

1. *Abstraction.* The set of features of SUT that are the subject of the testing activity are synthesized into a proper model.
2. *Test Generation.* Tests are produced from the model accordingly with a specified coverage criterion. Typically, the test generation phase ends when the set of generated tests (i.e., test suite) which satisfy the input coverage criteria. In this phase, tests are abstract and cannot be executed on the SUT.
3. *Concretization.* Abstract tests are converted into concrete/executable tests that can be executed on the SUT by an autonomous agent.
4. *Execution.* Concrete test cases are executed on the actual SUT. In this phase, it is possible to collect code coverage data as well as expose faults in the SUT.

Modeling the SUT as an Extended Finite State Machine

Different modeling approaches and languages are available depending on the nature of the SUT and the desired testing objective. EvoMBT uses extended finite state machines (EFSMs) as a modelling tool for capturing the desired behavior of the SUT. EFSMs are suitable for test generation of stateful applications (such as XR systems) as they provide internal variables that capture desired attributes and eventually use them to decide whether or not an action could be performed.

EFSMs are formal models where a system is represented by a number of states in which it can be at a given point, and changes from one state (source) to another (target) by means of transitions. Such transitions are guarded by conditions that depend on the internal variables of the model as well as input variables. The transitions could also have effects where they update the values of one or more internal variables. Two transitions are sequential if the target state of one transition is the source state of the other transition. A path on an EFSM is a finite sequence of sequential transitions from the initial state. A path is feasible if, when executed on the model, all the transition guards are satisfied. An abstract test case is a feasible path.

Concrete test cases as iv4XR framework Goal Structures

EvoMBT is integrated into iv4XR framework (see the iv4XR framework documentation[9]) and translates each transition into an iv4XR Goal Structure (see D2.4 for a detailed description of the framework). Concretized test cases can be then executed by the iv4XR agent as described in D2.4. Concretization and execution are specific for a system and EvoMBT exposes API to easily integrate SUT specific execution with model-based test case generation.

4.1.2 EvoMBT software architecture

EvoMBT implements 5 main components that are glued together via clearly defined interfaces and with a Main class that serves as a point of entry by exposing the various parameters of EvoMBT for command line as well as API access.

- *Model representation*. This is a core component that provides the representation formalism of EFSMs and all the corresponding operations on them.
- *Test case representation*. This component handles the representation of abstract test cases and test suites for use by the search algorithm. Test cases are feasible paths in the EFSM and test suites are sets of test cases. This component also provides different implementations of test case factories that are used for initializing the search algorithms with initial candidate test cases.
- *Coverage goals*. This component handles the representation of the various coverage goals. EvoMBT provides implementations for state, transition, and k-transition. Further coverage goals could easily be implemented by extending existing ones or by providing new implementations to the generic interfaces defined in this component.
- *Test case execution*. This component handles execution of abstract test cases on the model for the purpose of computing the corresponding fitness value, as well as the concretisation and execution of test cases on the SUT, whenever available. An execution of an abstract test case corresponds to replaying the path represented in the test case on the model, starting from its initial state. During execution, different observers are notified of different events, e.g., a transition traversed. One such observer is the coverage goal manager that deals with keeping track of the execution trace and calculation of fitness values. EvoMBT natively supports concretization for Lab Recruits and Space Engineers (see Section 4.1.3 below).
- *Algorithm*. This component is responsible for the various algorithms used for test generation. EvoMBT uses the algorithms implemented in EvoSuite. Since the interfaces used in EvoMBT are compatible with those in EvoSuite, most of the algorithms implemented in EvoSuite are readily usable in EvoMBT. Whenever there are strong deviations in some algorithms, they can be adapted for EvoMBT accordingly.

The EvoMBT tool is presented in [FR+22]. Instructions for running and extending EvoMBT are available in the EvoMBT wiki.

[9] https://github.com/iv4xr-project/iv4xr-framework

4.1.3 Case Studies

EvoMBT is generic and not bound to a specific SUT. In the context of the project, we implemented native support for two games: Lab Recruits and Space Engineers.

Lab Recruits

Lab Recruits[10] is a 3D game developed in the context of the iv4XR project. Here, we recapitulate the main features relevant for MBT. Lab Recruits game levels consist of different maze-like scenarios involving one or more players interacting with surrounding entities and with each other. The game levels are created by the game designer through a customized level notation. The level designs are saved as comma separated value (csv) files and loaded into the game by the player. To describe how EvoMBT applies to Lab Recruits, we consider the level in **Figure 4.2** that extends the level presented in **Figure 1.4**. The level in **Figure 4.2** includes three doors (door1, door2, and door3), four buttons (b1, b2, b3, and b4), and one goal flag (gf0). Doors are opened/closed by pressing on one or more buttons. It is however not necessary that all buttons be connected to doors, i.e., some buttons may not be connected to any door. In the level of **Figure 4.2**, b0 is not connected to any door, hence pressing it has no effect on the status of the doors in the level. Goal flags award points to the player and can be used to simulate treasures or game rewards. The player has to open door3 and reach goal flag gf0. To do so, the player needs to press a number of buttons in specific sequences. One possible sequence of actions could be to press first b1 to open door1, then b2 to open door3, however since b2 toggles door1 as well, the player requires to open door1. The player needs to pass door2 and press b3 to open door1. At this point the player can go to door3 and reach gf0. We can see from this example that even if the level seems simple, activating certain aspects of the level is non trivial, in particular from the perspective of automated testing. Clearly when the levels become large and complex, the difficulty increases accordingly.

Figure 4.2: Lab Recruits level extension of **Figure 1.7**

Figure 4.3 shows one possible way of modeling the Lab Recruits level in **Figure 4.2**. The model captures the essential features of the level while abstracting away from details that are not of

[10] https://github.com/iv4xr-project/labrecruits

interest to the tester, such as the precise position in the 3D world. The model focuses on checking the consistency of the button-door connections as well as the correct behavior of goal flags.
In general, game entities, such as buttons, goal flags, and each side of a door, are represented as EFSM states. For each door, states d_p and d_m model the two sides of door d. The EFSM in **Figure 4.3** has four button states, b0, b1, b2, and b3, six door states d1m, d1p, d2m, d2p, d3m, and d3p, and a goal flag state gf0.

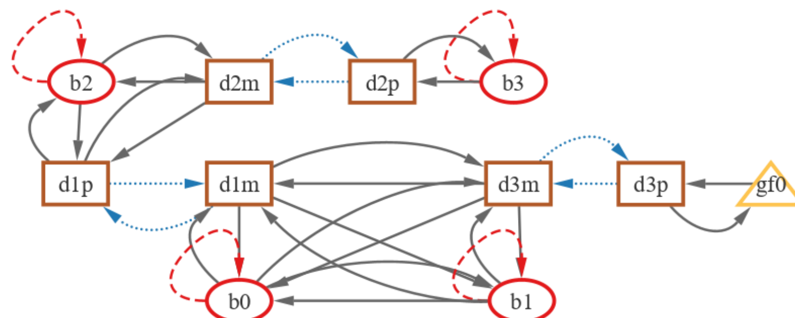


Figure 4.3: EFSM model for Lab Recruits level in **Figure 4.2**.

EFSM internal variables record door status. The EFMS has a boolean variable for each door that is true if the door is open and false otherwise.
As stated above, EFSM transitions represent player actions. With the abstraction of the model in **Figure 4.3**, a player can move from an entity to another, walk through a door, and toggle a button. Therefore, the EFSM has three types of transitions: solid edges for free travel, when the agent can move from one entity to the other without traversing a door; this type of transition has empty guard and effect. Dotted transitions model guarded movements that happen when the agent walks through a door; the guard checks the status of the corresponding internal variable. Dashed self loop transitions are for toggle actions, i.e, the agent presses the button; the effect changes the value of the internal variables associated with the doors connected to the pressed button.

Running EvoMBT on Lab Recruits

In this section, we will report the results of a small experiment in which EvoMBT is applied to Lab Recruits level. A more extensive study is presented in [FR+21].
To challenge the search algorithms included in EvoMBT, we consider a LabRecruits level, named Large, significantly larger than the one in **Figure 4.2**. as it includes 50 states, 194 transitions, and 15 internal variables. We consider five different test generation strategies included in EvoMBT: MOSA, MONOTONIC, STEADY STATE, SPEA2, and NSGAII. As the considered generation strategies are inherently stochastic, we run EvoMBT on the large model 20 times for each test generation strategy. Each run has a time budget of 300s, namely, the test generation process stops after 300s if the coverage is less than 100%. **Figure 4.4** summarizes the results of the experiments plotting the coverage over time achieved by each test generation strategy.

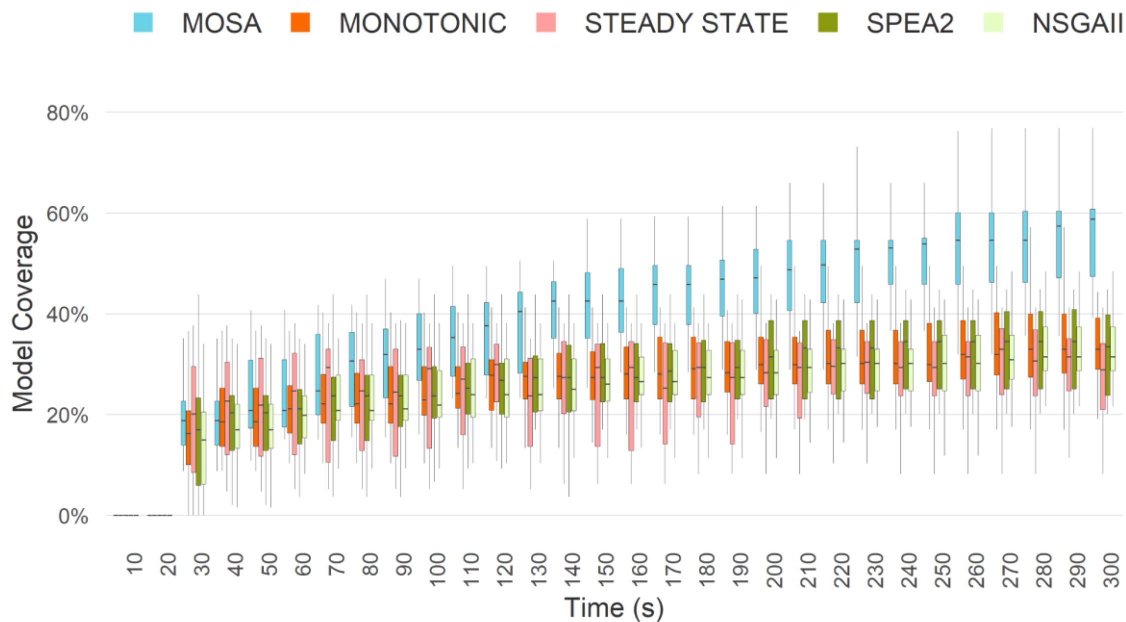


Figure 4.4: Model coverage over time for a Lab Recruits level.

The experiment highlights the main features of EvoMBT when executed on a model. First of all, EvoMBT allows using different test generation strategies that may have different performance. In this experiment, MOSA strategy outperforms other methods in terms of both effectiveness, i.e. the final coverage achieved, and efficiency, the convergence speed. However, different models could give different results and EvoMBT gives the testers the flexibility to select the approach more suited for the actual SUT. Moreover, EvoMBT reports detailed information about execution. In particular, EvoMBT takes a snapshot every *t* seconds, where *t* is a user-defined parameter (10s in **Figure 4.4**), and collects different information including the number of covered goals, the coverage, the number of fitness evaluations, and the percentage of the feasible path. All this information informs the tester about the generation process and helps in refining the model definition.

Finally, we concretize and execute abstract test cases generated on Lab Recruits level Large. EvoMBT produces a high-level report that includes the number of tests executed/passed and the total time required. Moreover, EvoMBT creates a detailed summary of test execution that includes detailed information about each executed iv4XR Goal Structure. A detailed description of EvoMBT is available in the wiki.

Space Engineers

The Space Engineers (SE) game is one of the industrial use cases of the iv4XR project. The SE plugin developed within the framework exposes the API used by EvoMBT. Therefore, it is possible to load a csv level and create the corresponding maze level into the SE environment.

Similarly to LabRecruits, a maze level consists of several rooms connected by doors, which supports the definition of Extended Finite State Machines (EFSM) and the automatic generation of test cases. A detailed description of this EvoMBT approach for Space Engineers is reported in D5.4.

4.2. Quality-Diversity Optimisation for Testing Space Engineers

In the game Space Engineers, one to the pilots of the iv4XR project, has many different types of blocks, which can be placed and interacted with in many different ways. The placement of blocks next to one another can also change their properties, which makes the task of testing the interactions with these blocks a complex one. One of the biggest problems is ensuring relevant coverage of the interaction space when the amount of blocks and their possible combinations makes it infeasible to test every scenario. One particular situation that is important to test is the interaction of multiple users with the same blocks, given the multiplayer gameplay of SE.

To tackle this problem, we developed a tool that generates test cases composed of actions for two test agents that promotes diversity for groups of sequences of actions in a given Space Engineers scenario. We created simulated versions of Space Engineer's levels based on grids and block ids, as can be seen in Fig. 4.5. The map is always a 2D square and the blocks are always connected to each other. Just like the Space Engineers game, blocks can be armor blocks or functional blocks, depending on their id. Blocks can be “reachable” or “unreachable” depending on whether they have a visible side or not. Agents can only interact with reachable blocks.

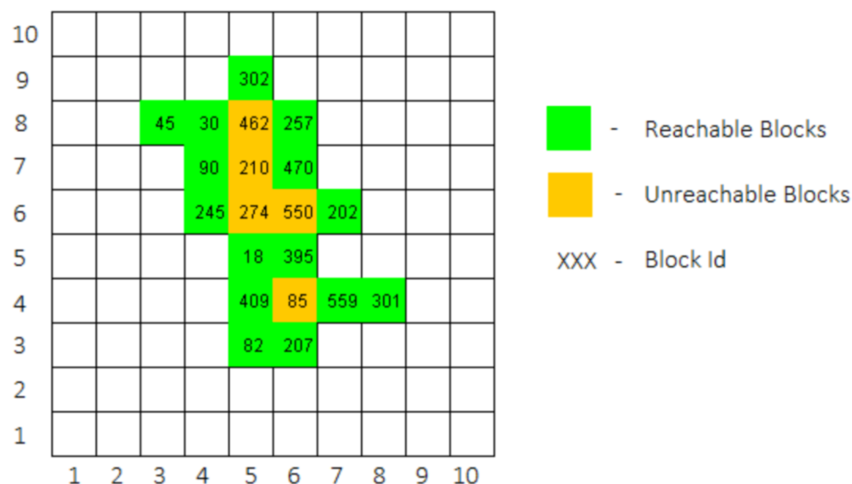


Figure 4.5: A Space Engineers game map represented as a grid. In this example, the map dimensions are 10x10 with 20 total blocks in it.

By implementing a version of the Quality-Diversity optimisation algorithm to generate grids of action sequences, our tool has shown to be capable of creating grids with very good total diversity values, ensuring that various interactions are covered and that redundant testing is minimized. Actions are defined by the interaction performed (grinding, welding, or using) and what block is the target of the action. Actions are characterized by a code where the interaction is represented

by its first letter (“G”, “W” or “U”) and the target block is defined by its position coordinates of the square map. For example, the code “W-0,1” means applying the action “welding” to the block at the position “0,1”. We decided to diversify action sequences instead of standalone actions. So, we will be generating action sequences for each player: as there are two players, we refer to them as an action sequence pair.

The diversity is guided by a 2D Quality-Diversity grid, whose dimensions are “Action Repetition”, and “Block Repetition”. Therefore, the tool generates diverse sequences of actions for the two agents that differ in the repetition of actions (performing the same actions repetitively or avoid repeating the same action) and differ in the block targeted (both agents using the same block, or using different ones). An example of a 5x5 grid and an example of action sequence pairs in two different positions can be found in Fig. 4.6.

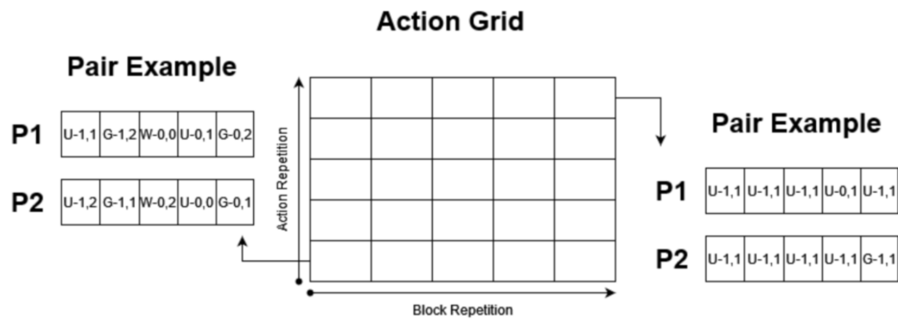


Figure 4.6: Action Grid Example: a 5x5 grid with examples of the pairs in positions (0,0) and (4,4).

In order to evaluate the bug detection capabilities of this tool, we created a game simulator, accompanied by a bug generator, where the actions from the grid are performed and generate the expected outcomes that would be generated in the real game. Our results showed that the tool is capable of generating test cases that can detect a good number of single-player bugs as well as multiplayer bugs. The comparison of the approach versus the use of randomly chosen action sequence pairs is shown in Figure 4.7.

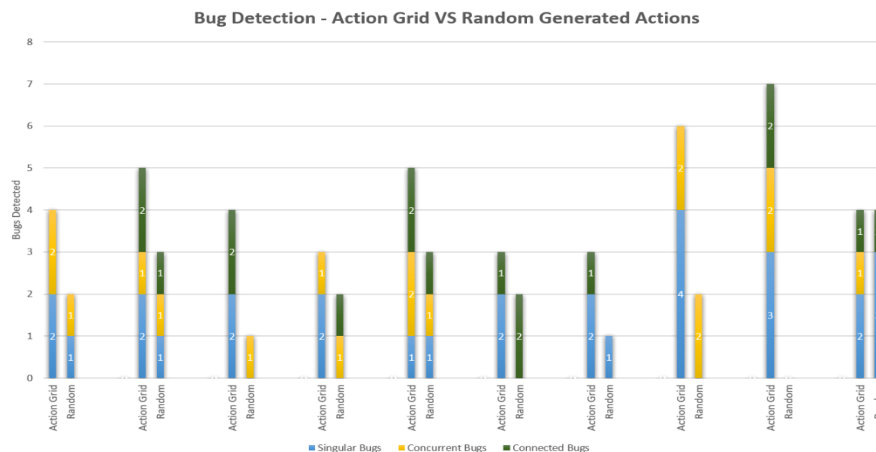


Figure 4.7: Chart showing the results of bug detection of the action grids approach vs the use of random sequence pairs.

5. Augmented Reality

5.1. Introduction

Taking as a reference the Google ARCore project, capable of creating Augmented Reality experiences, we developed a new tool that uses this technology, and implements tests that evaluate properties such as the position and size of AR objects in AR environments. The core approach was using the Record and Playback functionalities in order to support recording AR sessions and run the AR tests on these recordings.

The recorded AR sessions are used as inputs in the tool, which allow to establish desirable test environments. These represent common AR interactions that include moving the mobile phone camera and showing some real world objects. The tests make it possible to verify that certain properties of AR objects are met in the recorded environment.

5.2. Use Case

To demonstrate the tool we developed a simple AR application as the System Under Test. This SUT is a mobile application, the game Tanky, that uses the camera input to allow users to capture the real environment and put virtual 3D objects on it. The application is able to analyze the real environment and infer the available surfaces. Based on that it defines appropriate coordinate axes to place the virtual 3D objects.

Tanky

Tanky is an AR application that implements a simulation of a conflict between two war tanks. This tank game allows the user to interact with virtual tanks and place them in the real world. With the mobile device, it is possible to capture the real scene of the environment and tap the screen to insert a tank. A 3D object with the shape of a tank is placed in the location defined by a 3D point (x, y, z), inferred from the context. Similarly, more tanks can be added to the environment by tapping the screen. This will cause each one to occupy a certain place and, persistently, retain its location properties throughout the AR session.

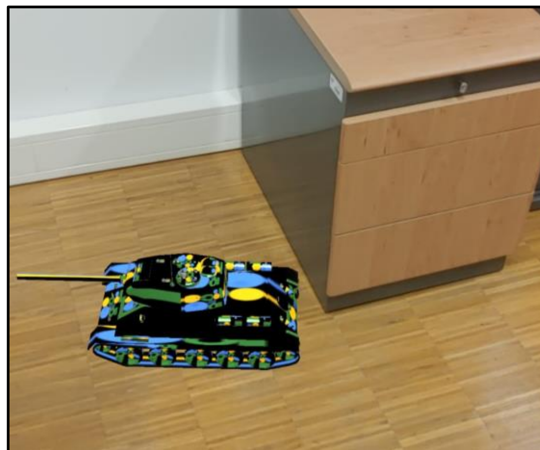


Figure 5.1: Virtual tank in a real scenario using Tanky

The game is played in turns. On the first turn, tapping the screen will place a tank in the AR world; in the next turn, another tank will be added. The next actions of touching the screen will make the tanks shoot, alternating each turn.

Google ARCore

Tanky uses the features of ARCore, Google's open source platform for creating augmented reality experiences. Through the mobile device's camera, ARCore is able to integrate virtual elements with the real world. It does this by tracking the mobile device as it moves, creating its own perception of the real world.

We used the ARCore *Record* and *Playback* functionalities to implement the testing tool.

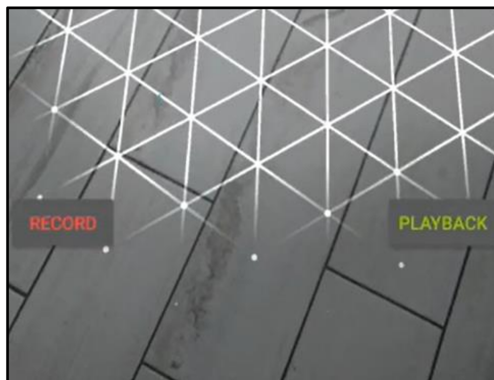


Figure 5.2: Record and Playback buttons in the AR application

Record

First, we recorded a set of videos that constitute a gallery of situations (e.g. test cases) to use in the AR application under test. The recordings are sessions stored on the mobile device containing specific metadata that will allow the simulation of the user interacting with the recorded session later, in a playback session. We recorded videos with distinct characteristics combining still, moving, rotating the camera device, with moving back and forth relative to a specific point, or circling it The set can be extended by the tester to include different testing situations in the gallery.

Playback

It is possible to navigate and open the folder containing the gallery of previously recorded videos. In the playback, the user is a spectator and the system will interpret the recorded physical environment and detect the different surfaces and the coordinate system. The environment will be ready for interaction after this step, e.g., 3D tanks can be placed on the surfaces detected by the application.

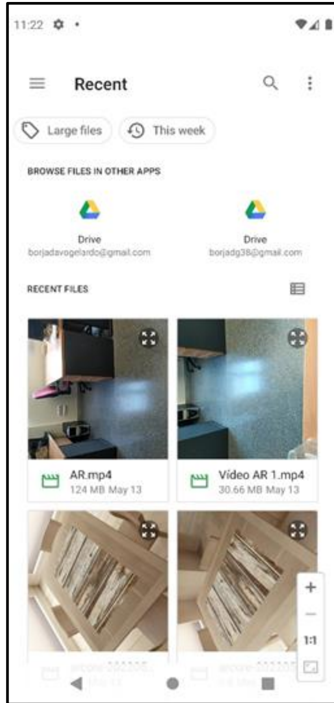


Figure 5.3: Gallery of recorded AR sessions

5.3. The testing process

The components involved in the AR test process and are related to the SUT are depicted in figure 5.4.

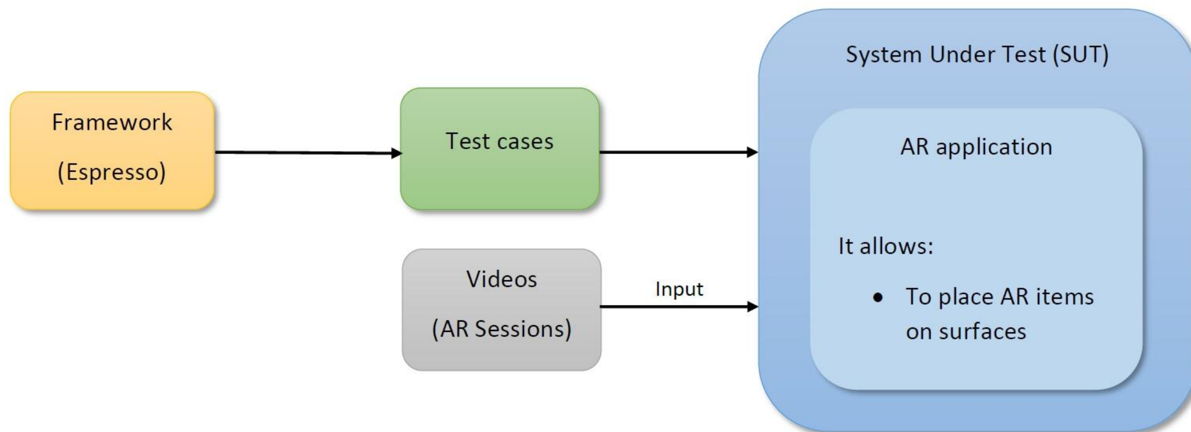


Figure 5.4: Elements of the AR testing process

Espresso framework

The Espresso testing framework is used for the construction of the tests in a quick and effective way. This technology is intuitive and easy to read, and is key to supporting automated testing.

To create an Espresso test it is necessary to use the AndroidJUnit4 class, declare the *@Rule* and write the *@Test* (see figure 5.5).

```
@LargeTest
@RunWith(AndroidJUnit4.class)
public class SurfaceTest {

    @Rule
    public ActivityTestRule<HelloArActivity> mActivityTestRule =
            new ActivityTestRule<>(HelloArActivity.class);

    @Test
    public void helloArActivityTest() throws InterruptedException {
```

Figure 5.5: Use of JUnit, @Rule and @Test for the design of a test

The *@Rule* references the main class. And inside *@Test*, all the code that represents the logic of the test is written. An example of Espresso-based code that produces a button click (on the Playback button) is shown in figure 5.6.

```
ViewInteraction appCompatButton = onView(
        allOf(withId(R.id.playback_button), withText("Playback"),
                childAtPosition(
                        childAtPosition(
                                withId(android.R.id.content),
                                position: 0),
                        position: 3),
                isDisplayed()));
appCompatButton.perform(click());
```

Figure 5.6: Click to Playback button using Espresso

Inputs

For the design of specific test cases, it has been sought to use certain recorded AR sessions, which allow the test cases to be addressed in a more effective way. In the elaboration of the test cases, possible relevant behaviors of the AR technology that can be tested and are relevant have been analyzed.

For a specific test, it will be needed to record a specific AR session. Multiple tests can use the same recorded video, depending on the test objectives.

The test needs pre recorded videos of AR sessions, as stated before. These videos must have good lighting, and the colors of the environment must not be too similar to each other. This will make it easier to automatically detect surfaces and detectable elements in the application.

Outputs

The tester will receive the appropriate feedback if any of the conditions and assertions of a test are not met. The tool will present a message to the user indicating that the SUT has not passed the test. The letters of the text are red, with a saturation and contrast in harmony with the background.

On the contrary, if the test process has ended without any problem, the message that will be displayed to the user will contain the text "Test passed". In this case, green tones are used which, as in the previous case, are balanced with the general color range.

In addition, the system output, reflected in the Run tab, will indicate if the test has been passed. Otherwise, it will return an error and give information about the part of the code associated with it.

Figure 5.7: Test Failed pop-up message

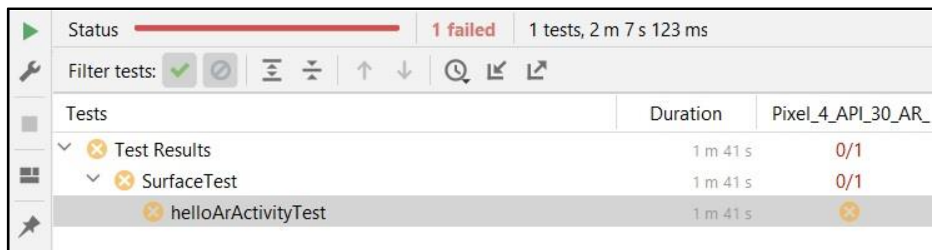


Figure 5.8: Failed test result

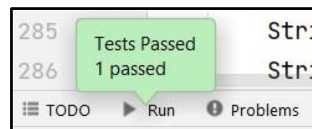


Figure 5.9: Test Passed pop-up message.

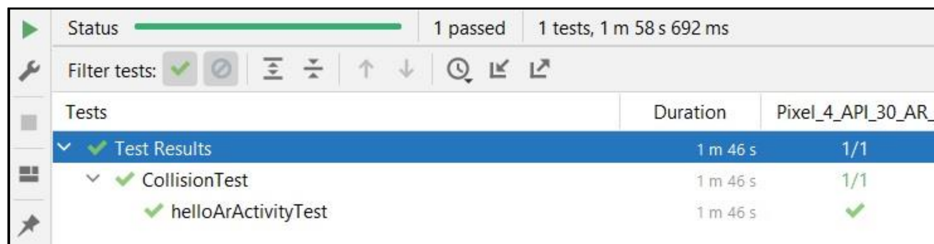


Figure 5.10: Passed test result

Figure 5.11: Final message that indicates if the test was passed or not

Test construction

For the elaboration of a test, in the first place one needs to determine what needs to be tested. This can be a property of a 3D object that exists in the AR environment, or how a 3D object interacts with the environment, among others.

Then, a process has been designed that represents the list of actions that the test agent will execute automatically in the SUT, to force a context in which the test can be done properly. This involves the following steps:

1. Load and play a previously saved video that refers to the AR sessions selected for the test. This can be achieved by:
    a. Pressing the "Playback" button to access previously saved videos in the test gallery.
    b. Select the target video and play it.
2. Carry out the actions on the AR session, such as placing 3D objects in the environment.

After automatically performing the necessary actions on the AR session, assertions are executed in the test, to determine if certain rules that establish that the test has been passed successfully are met. A message shows the results, using green letters, informing success, or using red letters to indicate failure.

Test execution

To run the tool on a physical mobile device, it must first be configured to allow *USB debugging*. To do this, the user has to navigate to *USB debugging* (*Settings* > *Developer options* > *USB debugging*) and enable it. Then the device must be connected to the computer where the tool project is located. Finally, when the device is detected in Android Studio, the user must click on the *Run* button.

To run an AR test, users must first load the AR test package called *ar_tests*. In the project structure, the user will see the set of tests available.

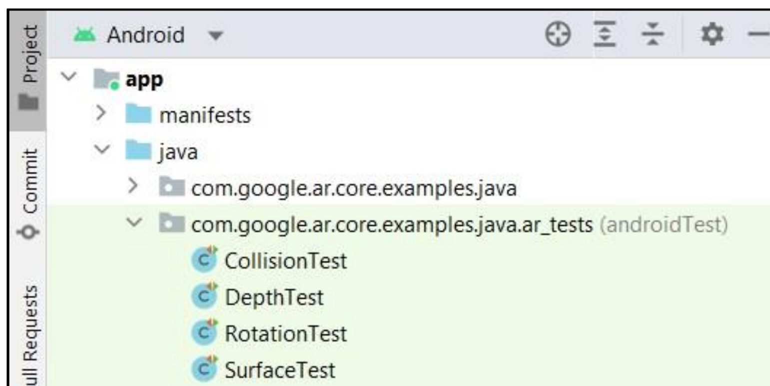


Figure 5.12: Location of AR tests in the project

To run one of the tests, these steps can be followed alternatively:

- Right click on the test > Click on *Run*.
- Double click on a test to open its code > Click on the *Run* button, represented as a green triangle on the toolbar.
- Double click on a test to open its code > Click on the *Run* tab > Click on *Run*.

It is convenient to make sure that the name of the desired test appears as selected, so that it is the one that is executed. That can be checked on the toolbar.

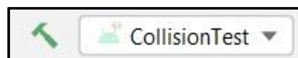


Figure 5.13: Selection of activity to be executed.

The selected test will then be executed, automatically performing the actions specified in its code, as well as the corresponding checks and assertions.

5.4. INTEGRATION WITH THE IV4XR FRAMEWORK

Regarding the integration with the iv4XR framework, the needed libraries (e.g., *aplib*) were added to the AR testing project. Then some java classes were created to adapt the application to the iv4XR architecture. These classes are *MyAgentEnv*, *GoalLib* and *MyAgentState*. A test file will use these resources to establish the operation of a test designed with iv4XR features.

MyAgentEnv

MyAgentEnv creates an environment based on the AR application activity. It also contains methods like *observe*, *tapScreen*, *clickButton* and *selectVideo*, allowing these last three to perform actions in the environment.

GoalLib

This class contains methods to create goals based on the possible actions specified on *MyAgentEnv*.

MyAgentState

MyAgentState extends *Iv4xrAgentState*, it implements the method *updateState*, responsible for updating the state based on the anchors displayed.

Test files

Each test file contains a goal structure in which there is a sequence of goals, being these related to clicking the Playback button, selecting a recorded AR session and then tapping the screen to place AR objects. After that, there are assertions that determine whether the test passes, based on certain criteria.

In a test, the necessary goals are grouped in a GoalStructure, which represents the sequence of actions that must be executed in the system automatically. An example of a GoalStructure would

consist of the goals related to clicking the Playback button, selecting a video from the gallery and touching the screen.

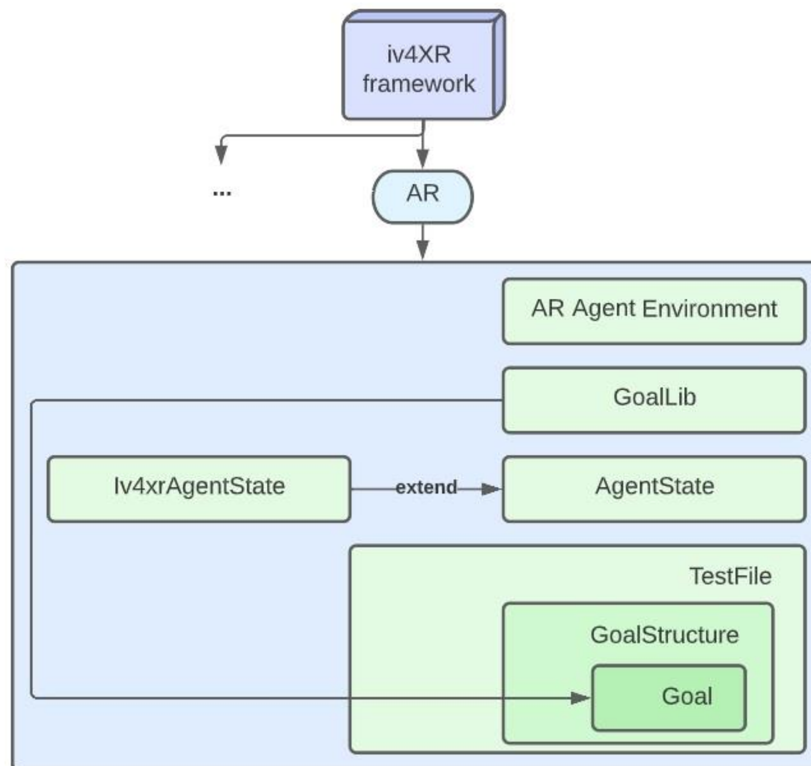


Figure 5.14: Architecture of AR test system integrated with iv4XR framework

5.5. AR TESTS

During the development of the game, several bugs were included on purpose to simulate incorrect scenarios and thus validate our testing methodology.

Four tests were designed and implemented to verify different properties in AR environments. They use Tanky as the SUT, and automatically execute actions in the environments it provides.

First of all, the Surface Test focuses on ensuring individual properties of each tank. Secondly, the Collision Test looks for the validation of behaviors that involve more than one tank at a time. Third, the Depth Test aims to verify the capabilities of the application's depth system, checking the ARCore depth API. Finally, the Rotation Test is aimed at analyzing the state of the tanks when the user changes the perspective of the mobile device. We tested, additionally, if the maximum number of tanks that can be placed is 2.

Surface test

At the start of each game, two tanks must be placed in the environment, representing the two characters in the game. The tanks must appear in the place where the user has decided by touching the screen, and the properties with the X, Y and Z axes must be respected. In this case,

it will be tested that, when placing a tank, the only rotation assigned to it is related to the Y axis. This assures that the tank is correctly placed on the surface.

When the tank is placed in the environment it must not have rotation in the X or Z axes, which would imply that it is not well positioned on the surface. If the X or Z axis rotation value is non-zero, the tank will be oblique to the surface, which should not be correct. In that case, the test will fail.

In terms of development, a GoalStructure was created in the test file that prepared the test scenario, as can be seen in the following image.

```
GoalLib goalLib = new GoalLib() ;
GoalStructure G = SEQ(
        goalLib.clickButtonG(agent, buttonName: "Playback", sleep: 2000),
        goalLib.selectVideoG(agent, videoPosition: 1, sleep: 35000),
        goalLib.tapScreenG(agent, x: 300, y: 1500, sleep: 3000),
        goalLib.tapScreenG(agent, x: 600, y: 1500, sleep: 3000),
        goalLib.tapScreenG(agent, x: 400, y: 1000, sleep: 3000),
        goalLib.tapScreenG(agent, x: 500, y: 1000, sleep: 5000)
) ;
agent.setGoal(G) ;
```

Figure 5.15: Sequence of goals specified in an AR test

Then, an assertion is used to check that for each tank, the value of its rotation in the X and Z axes is zero. If so, a *Test Passed* message is returned, and the assertion causes test results returned at the system-level to be *Passed*.

```
boolean surfaceCondition = ((float) a.properties.get("qx") == 0.0) &&
        ((float) a.properties.get("qz") == 0.0);
if(!surfaceCondition) {
    mActivityTestRule.getActivity().testFinishedMessage( passed: false);
    Thread.sleep( millis: 60000);
}
assertTrue(surfaceCondition) ;
```

Figure 5.16: Assertion for the surface test

Collision test

In the real world, tanks cannot occupy regions of common space. Each one is located in a place, and a point located in the coordinates X, Y and Z can correspond to the space occupied by a tank or by none; never by two tanks.

This must be true in the application. Two virtual tanks cannot be in the same place at the same time. Similarly, if these tanks change position over time, they should not be able to intersect.

To achieve this, the dimensions of the virtual tanks have been determined. These represent the associated hitbox. First, the virtual object file of the tank has been processed, and its lines have been iterated to find those that start with v and represent the vertices of the tank. In each of these

lines three sections can be distinguished, each of them representing a coordinate on the X, Y and Z axis, respectively. From this information, the maximum and minimum value relative to each axis has been calculated: *max(X), min(X), max(Y), min(Y), max(Z), min(Z).*

```
# Blender v2.82 (sub 7) OBJ File: 'pawn.blend'
# www.blender.org
mtllib pawn.mtl
o pawn
v -0.050448 0.113797 0.000000
v -0.049923 0.114325 -0.004917
v -0.049201 0.114325 -0.009787
v -0.048005 0.114325 -0.014562
v -0.046346 0.114325 -0.019197
```

Figure 5.17: 3D tank vertex data

Once these values have been calculated, the length of the tank on an axis is calculated as the subtraction between the maximum and minimum value for that axis:

$$length(X) = max(X) - min(X)$$

$$length(Y) = max(Y) - min(Y)$$

$$length(Z) = max(Z) - min(Z)$$

The next step is to know the location of a virtual tank that has been placed in the real world. This information corresponds to the translation values of the virtual object:

$$location(tank, X) = translation(tank, X)$$

$$location(tank, Y) = translation(tank, Y)$$

$$location(tank, Y) = translation(tank, Y)$$

With the location and size of the tank its hitbox can be computed. The hitbox is centered on the middle of the 3D model the represented the tank:

$$hitbox(tank, X) = (location(tank, X) - length(X)/2) .. (location(tank, X) + length(X)/2)$$

$$hitbox(tank, Y) = (location(tank, Y) - length(Y)/2) .. (location(tank, Y) + length(Y)/2)$$

$$hitbox(tank, Z) = (location(tank, Z) - length(Z)/2) .. (location(tank, Z) + length(Z)/2)$$

By checking if the tanks' hitboxes intersect we can test the occurrence of a collision. In the case that this is not the intended behavior, the test should fail.

Depth test

ARCore's Depth API allows users to estimate depth from real-world information. The shape and size of the elements captured by the camera serve as support to create a depth image every time. These images represent areas with associated colors, with the red areas being the closest ones, and the blue areas being the furthest away areas.

As it is typical in a war-fighting game, barriers and hiding spots should be used to protect the tank but also constitute obstacles for its movement. The obstacles/hiding locations in this case are real-world items, so it is required to map them and understand the distances between them and the camera. This supports checking the occlusion of objects in the environment (i.e, determine whether the tank should be positioned in front of or behind the real object). To achieve this, the tool must verify if the depth image generated by the application corresponds to reality.

```
if (camera.getTrackingState() == TrackingState.TRACKING
        && (depthSettings.useDepthForOcclusion()
        || depthSettings.depthColorVisualizationEnabled())) {
  try (Image depthIm = frame.acquireDepthImage16Bits()) {
    backgroundRenderer.updateCameraDepthTexture(depthIm);
    depthImage = depthIm;
```

Figure 5.18: Depth image update

First, the depth image is obtained at a given time: *depthImage(time).* In addition, two areas of the scene must be known in which, at said instant of time, they have different depths.

Next, the color of a pixel located in the area of the depth image that should be closer is obtained, and also the color of another pixel in the area that should be further away: *color(nearPixel), color(farPixel)*.

Finally, through calculations, it is determined if the color of the pixels implies that the depth of the pixel of the far zone is greater than the depth of the pixel of the near zone.

```
if (nearPredominantColor == "red" && farPredominantColor == "red") {
    if (nearRed != farRed) {
        if (nearRed > farRed) correctDepth = true;
    }
} else if (nearPredominantColor == "red" && farPredominantColor == "green") {
    correctDepth = true;
} else if (nearPredominantColor == "red" && farPredominantColor == "blue") {
    correctDepth = true;
} else if (nearPredominantColor == "green" && farPredominantColor == "green") {
    if (nearGreen != farGreen) {
        if (nearGreen > farGreen) correctDepth = true;
    }
} else if (nearPredominantColor == "green" && farPredominantColor == "blue") {
```

Figure 5.19: Comparison of the depth of two points

Rotation test

This game is an AR application in which the map can be the entire environment through 360 degrees. The virtual tanks position should be consistent as they must stay in the same place even if the mobile device's camera has turned the other way and the tank is no longer visible. If the mobile moves away from a virtual object, when the user returns the focus of the camera to the place where the virtual object was, it should continue to appear there.

To analyze this behavior, a video has been provided that works as input, in which the device manages to rotate and leave possible virtual objects placed in the scene out of focus.

At the code level, when putting a virtual tank in the real world, the *TrackingState* property of the element has been analyzed. The value of this property can be *Paused*, *Stopped*, or *Tracking*, depending on whether the *Trackable* is currently tracked. This gives us the information that a virtual object is in the scene. Next, a certain millisecond wait is performed, after which the virtual tank will have been left out of focus in the AR session being played.

Finally, the value of this property is evaluated again, and it is verified that it is equal to the value obtained previously. Then it is verified that this value is Tracking, which would show us that the virtual object still retains its properties and continues to exist even if the camera is not pointing at it.

```
if (trackingStateObjectSeen == trackingStateObjectNotSeen) {
    if (trackingStateObjectSeen == "TRACKING") {
        correctRotation = true;
    }
}
```

Figure 5.20: TrackingState check.

FTAs Scientific Publications

- Samira Shirzadehhajimahmood, Wishnu Prasetya, Frank Dignum, Mehdi Dastani, An Online Agent-based Search Approach in Automated Computer Game Testing with Model Construction. In Proceedings of the 13th International Workshop on Automating TEST Case Design, Selection, and Evaluation. 2022. doi.org/10.1145/3548659.3561309

- R. Ferdous, C. Hung, F. M. Kifetew, D. Prandi, A. Susi. *EvoMBT*. 15th IEEE/ACM International Workshop on Search-Based Software Testing (tool competition), SBST@ICSE 2022. doi:10.1145/3526072.3527534.

- Pastor Ricós, F. (2022). *Scriptless Testing for Extended Reality Systems.* In International Conference on Research Challenges in Information Science (pp. 786-794). Springer, Cham. doi.org/10.1007/978-3-031-05760-1_56

- Mulders, A., Valdes, O.R., Ricós, F.P., Aho, P., Marín, B., Vos, T.E.J. (2022). *State Model Inference Through the GUI Using Run-Time Test Generation.* In: Guizzardi, R., Ralyté, J., Franch, X. (eds) Research Challenges in Information Science. RCIS 2022. Lecture Notes in Business Information Processing, vol 446. Springer, Cham. doi.org/10.1007/978-3-031-05760-1_32

- I. S. W. B. Prasetya, Fernando Pastor Ricós, Fitsum Meshesha Kifetew, Davide Prandi, Samira Shirzadehhajimahmood, Tanja E. J. Vos, Premysl Paska, Karel Hovorka, Raihana Ferdous, Angelo Susi, and Joseph Davidson. 2022. *An agent-based approach to automated game testing: an experience report.* In Proceedings of the 13th International Workshop on Automating Test Case Design, Selection and Evaluation (A-TEST 2022). Association for Computing Machinery, New York, NY, USA, 1–8. doi.org/10.1145/3548659.3561305

- Ansari, Prasetya, Dastani, Dignum, Keller. An Appraisal Transition System for Event-Driven Emotions in Agent-Based Player Experience Testing. In International Workshop on Engineering Multi-Agent Systems (EMAS), 2021. doi.org/10.48550/arXiv.2105.05589

- R. Ferdous, F. M. Kifetew, D. Prandi, A. Susi. *Towards Agent-Based Testing of 3D Games using Reinforcement Learning.* 37th IEEE/ACM International Conference on Automated Software Engineering, ASE4Games 2022.

- R. Ferdous, F. M. Kifetew, D. Prandi, I. S. W. B. Prasetya, S. Shirzadehhajimahmood, A. Susi. *Search-based automated play testing of computer games: A model-based approach.* 13th International Symposium, SSBSE 2021. doi:10.1007/978-3-030-88106-1_5

- Vos, T. E., Aho, P., Pastor Ricos, F., Rodriguez Valdes, O., & Mulders, A. (2021). *testar–scriptless testing through graphical user interface.* Software Testing, Verification and Reliability, 31(3), e1771. doi.org/10.1002/stvr.1771

- Rodríguez-Valdés, O., Vos, T.E.J., Aho, P., Marín, B. (2021). 30 Years of Automated GUI Testing: A Bibliometric Analysis. In: Paiva, A.C.R., Cavalli, A.R., Ventura Martins, P., Pérez-Castillo, R. (eds) Quality of Information and Communications Technology. QUATIC 2021. Communications in Computer and Information Science, vol 1439. Springer, Cham. doi.org/10.1007/978-3-030-85347-1_34

- van der Brugge, A., Pastor-Ricós, F., Aho, P., Marín, B., & Vos, T. E. (2021). Evaluating TESTAR's effectiveness through code coverage. Actas de las XXV Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2021), 1-14.2021/JISBD/2021-JISBD-042.pdf

- Extended abstract: *Aplib: An Agent Programming Library for Testing Games*, I. S. W. B. Prasetya, Mehdi Dastani, in the International Conference on Autonomous Agents and Multiagent Systems (AAMAS), 2020.

- Concepts behind agent-based automated testing: Aplib: Tactical Agents for Testing Computer Games. I. S. W. B. Prasetya, Mehdi Dastani, Rui Prada, Tanja E. J. Vos, Frank Dignum, Fitsum Kifetew, in Engineering Multi-Agent Systems workshop (EMAS), 2020. doi.org/10.1007/978-3-030-66534-0_2

- Ricós, F.P., Aho, P., Vos, T., Boigues, I.T., Blasco, E.C., Martínez, H.M. (2020). Deploying TESTAR to Enable Remote Testing in an Industrial CI Pipeline: A Case-Based Evaluation. In: Margaria, T., Steffen, B. (eds) Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles. ISoLA 2020. Lecture Notes in Computer Science(), vol 12476. Springer, Cham. doi.org/10.1007/978-3-030-61362-4_31

- Thorn Jansen, Fernando Pastor Ricós, Yaping Luo, Kevin van der Vlist, Robbert van Dalen, Pekka Aho, and Tanja E. J. Vos, “Scriptless GUI testing on mobile applications”, QRS 2022, 22nd IEEE International Conference on Software Quality, Reliability, and Security.

REFERENCES

[AP11] VS Alagar and K Periyasamy. 2011. Extended finite state machine. In Specification of software systems. Springer, 105–128.

[BK08] Baier, Christel, and Joost-Pieter Katoen. Principles of model checking. MIT press, 2008.

[Ben08] Ben-Ari, Mordechai. Principles of the Spin model checker. Springer Science & Business Media, 2008.

[FR+21] Ferdous, R., Kifetew, F., Prandi, D., Prasetya, I. S. W. B., Shirzadehhajimahmood, S., & Susi, A. (2021, September). Search-Based Automated Play Testing of Computer Games: A Model-Based Approach. In Search-Based Software Engineering: 13th International Symposium, SSBSE 2021, Bari, Italy, October 11–12, 2021, Proceedings (pp. 56-71).

[FRK+22] Ferdous, R., Kifetew, F., Prandi, D., & Susi, A. (2022, October). Towards Agent-Based Testing of 3D Games using Reinforcement Learning. In Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, ASE4Games 2022, Rochester, MI, USA, October 10–14, 2022.

[FR+22] Ferdous, Raihana, Chia-kang Hung, Fitsum Kifetew, Davide Prandi, and Angelo Susi. "EvoMBT at the SBST 2022 Tool Competition." In 2022 IEEE/ACM 15th International Workshop on Search-Based Software Testing (SBST), pp. 51-52. IEEE, 2022.

[Shi+21] Samira Shirzadehhajimahmood, ISWB Prasetya, Frank Dignum, Mehdi Dastani, and Gabriele Keller. Using an agent-based approach for robust automated testing of computer games. In Proceedings of the 12th International Workshop on Automating TEST Case Design, Selection, and Evaluation. 2021.

[Shi+22] Samira Shirzadehhajimahmood, Wishnu Prasetya, Frank Dignum, Mehdi Dastani, An Online Agent-based Search Approach in Automated Computer Game Testing with Model Construction. In Proceedings of the 13th International Workshop on Automating TEST Case Design, Selection, and Evaluation. 2022.

[TV+21] Tanja Vos, Pekka Aho, Fernando Pastor Ricós, Olivia Rodriguez-Valdes, and Ad Mulders. testar–scriptless testing through graphical user interface. In Software Testing, Verification and Reliability. 2021.

[TP+22] Pierrot, Thomas, et al. "Diversity Policy Gradient for Sample Efficient Quality-Diversity Optimization." ICLR Workshop on Agent Learning in Open-Endedness. 2022.