Intelligent Verification/Validation for XR Based Systems

Research and Innovation Action

Grant agreement no.: 856716

D2.4 – Report describing the iv4xr Framework

iv4XR – WP2 – D2.4

Version 2.2

December 2022

Project Reference	EU H2020-ICT-2018-3 - 856716
Due Date	31/12/2022
Actual Date	28/12/2022
Document Author/s	Wishnu Prasetya (UU)
Version	2.2
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	29/11/2022	Initial document structure and contents	WP
1.1	11/12/2022	Writing up the first 7 sections.	WP
1.2	12/12/2022	Minor edits and corrections	RP
1.3	15/12/2022	Adding subsection about Report	WP
2.0	16/12/2022	Full draft	WP
2.1	23/12/2022	Addressing QA comments	WP
2.2	28/12/2022	Final arrangements for submission	RP

Document Quality Control			
Version QA	**Date**	**Comments (and if appropriate reason for change)**	**Initials of QA Person**
2.0	20/12/2022	Comments and minor edits	FK

Document Authors and Quality Assurance Checks		
Author Initials	**Name of Author**	**Institution**
WP	Wishnu Prasetya	Utrecht University
RP	Rui Prada	INESC-ID
FK	Fitsum Kifetew	FBK

TABLE OF CONTENTS

Executive Summary 1

Section 1 - Introduction 1

Section 2 - Related Work 4

Section 3 - Agents 6

Section 4 - Tactics and Goals 10

Action 11

Tactic 12

Goal and Goal Structure 13

Section 5 - Specifying Assertions 14

Asserting reachability 15

Asserting invariant 15

Asserting dynamic property 16

Reporting 17

Section 6 - Dynamic Goals 18

Section 7 - Deployment Architecture 20

Section 8 - Experience 21

Section 9 - Conclusion 23

References 23

EXECUTIVE SUMMARY

This document describes the iv4xr Framework. It starts by giving an overview of the Framework, its top level architecture and its main components. The document then focuses on the Framework Core part, as the other main components are described in other Deliverables. In this document we will also focus on a more conceptual presentation of the Framework Core. The GitHub-pages of respectively the Framework and the Framework Core provide instructions and further documentation on their usage.

Document overview.

- Section 1 gives an introduction. Section 2 gives a brief overview of related work.
- Section 3 explains the execution model of iv4xr agents and some basics about their automated navigation and exploration feature.
- Section 4 explains the concept of tactics and goals. These are key concepts for iv4xr test agents.
- Section 5 presents different types of assertions/specifications that iv4xr agents can check and mentions the kind of output the test agents produce.
- Section 6 discusses dynamic goals, which are essential to build smarter test agents.
- Section 7 briefly discusses how to deploy iv4xr into the testing setup of some system under test.
- Section 8 presents some selected results of our study on agent-based tests produced by iv4xr.
- Finally, Section 9 gives a conclusion.

SECTION 1 - INTRODUCTION

With Extended Reality (XR) Systems we refer to software applications that involve the use of visual virtual worlds, ranging from computer games, 3D simulators, to virtual reality (VR) and augmented reality (AR) applications. XR systems are on the rise: the hardware is improving, there is a steady stream of innovations to feed the market, and there is push from companies like Meta that seek to popularize XR systems. This also means that XR systems are also becoming more complex, e.g., modern computer games and 3D simulators improve realism and user experience by allowing users to have fine grained control/interactions. A downside of this development is that it becomes increasingly difficult to test these systems.

For example, to test that an XR system would maintain a specified correctness invariant of a certain family of states, the tester will first need to operate the application to bring it to at least one such state. This often requires a long series of fine grained interactions with the system. Only then the tester can check if the said invariant does hold in that state. Such a test is hard, error-prone, and fragile to automate. Consequently, developers resort to slow, expensive, and biased manual testing. Considering that the industry is worth over 100 billion USD, speeding up testing by automating it is a need that cannot be ignored.

As indicated above, a common test related task is to bring the system under test (SUT) to a certain state of interest (we can call this a goal state), either because we want to check if the state is reachable and correct, or because we need to do a specific action on this state that is required for the given test scenario. In principle this task is a search problem: given a search space with a certain structure, find at least one element of the space satisfying certain properties. There are indeed algorithms for solving such a problem. However, in the context of XR, the problem is very challenging. For example a computer game often employs randomness and it often consists of many entities that interact with each other and with the user. Some interactions might be cooperative while others can be adversarial. These and other factors lead to a vast and fine grained search space which is hard to deal with using existing automated testing techniques such as search based [Min04], model based [Net+07], or symbolic [Ana+13,SA06]. At least, directly applying these techniques would be futile.

The key to handling such a space is to have an approach that enables the programming of domain reasoning to express which parts of the search space are actually relevant to consider, and likewise what kinds of plans (for reaching given goal states) are needed. This allows the underlying test engine to focus its search on the parts of the interaction and plan spaces that semantically matter. Iv4xr Framework offers such an approach. The Framework offers a set of automated testing tools, but these tools are based on an agent programming core. This has been a deliberate choice, as in agent-based approaches autonomous distributed planning and reasoning based interactions with environments are present as first class features.

The agent-based core of iv4xr Framework (which we will often refer to as the *Framework Core*) is a Java library for programming intelligent agents suitable for carrying out complex testing tasks. They can be used in conjunction with Java testing frameworks such as JUnit, e.g. to collect and manage test verdicts. Figure 1 shows a 3D game we use as a pilot where an iv4xr agent was used to automate test scenarios/tasks. The Framework Core features *BDI* (Belief-Desire-Intention [Her+17]) agents and adds a novel layer of *tactical* programming that provides an abstract way to exert control on agents' behavior. Declarative reasoning rules express when actions are allowed to execute. Although in theory just using reasoning is enough to find a solution (a plan that would solve the given goal state) if given infinite time, such an approach is not likely to be computationally efficient. For testing, this matters as no developers would want to wait for hours for their test to complete. The tactical layer allows developers to program an imperative control structure over the underlying reasoning-based behavior, allowing them to have greater control over the search process. So-called tactics can be defined to enable agents to choose and prioritize their short term actions and plans, whereas longer term strategies are expressed as so-called goal structures, specifying how a goal can be realized by choosing, prioritizing, sequencing, or repeating a set of subgoals.

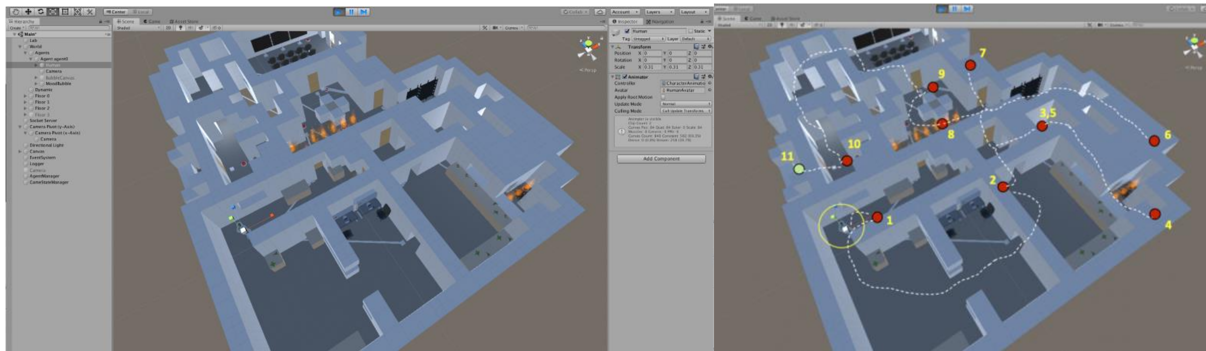


Fig. 1: *A game-level under development in Unity. Unity is a game engine, and it also provides a game development IDE, shown in the screenshot to the left. An example scenario/task to test is shown to the right, involving finding and interacting with 10 interactables (buttons) to reach an item that has a key role in the game level (marked green). The testing task is to verify that this key item is reachable. The task can be automated using iv4xr agent. So, each time the developers tweak the game-level's layout or logic they can just re-run the agent to check if the aforementioned key item is still reachable.*

As opposed to dedicated agent programming languages (e.g. JASON, 2APL, GOAL, SARL), iv4xr offers an embedded Domain Specific Language (DSL) of agent programming. The DSL is embedded in Java. Such an embedded DSL approach means that the agents are in principle programmed in Java, but the DSL provides a set of APIs that give the fluent appearance of a DSL. In principle, having a native programming language for writing tests is a huge benefit, but only if the language is rich enough and has enough tools and community support. Otherwise, it is a risk that most companies will be unwilling to take. On the other hand, using an embedded DSL means that the programmers have direct access to all the benefits of the host language, in this case Java: its expressiveness (OO, lambda-expression etc.), static typing, rich libraries, and a wealth of development tools. So overall, we believe that embedded DSL is the right approach to provide iv4xr technology for the industry.

While the agents themselves can do automated testing, they can also be used as a layer providing an abstract way to control and test a system under test (SUT). This makes it possible to build more test automation on top of the agents, including tools implementing aforementioned traditional automated testing approaches (e.g. search based testing, model based testing, etc). This allows them to target XR systems, while delegating the handling of tactical level control of the SUT to iv4xr agents.

In Figure 2 below we show the architecture of the overall system including the aforementioned iv4XR *Framework Core*. There are also four modules that are indicated as external modules. These are themselves tools for testing. They use the Core, but are managed outside the Core. The reason we do not include these modules in the Framework Core is twofold. First of all it emphasizes the fact that other modules can be added here for specific purposes. The second reason is that the software in which these modules are developed does not seamlessly connect to the Core. Each of the modules uses a different type of technique which are best supported and developed in a specific software environment. Within the project we had to make a trade off

between rapid and deep development of the technique in the appropriate software and integration into the complete architecture. As scientifically it is more interesting to further develop each of the modules we chose to use the most appropriate software environment for each of them at the cost of full integration. This does not mean that there is no integration at all, but just that the user needs to assemble the appropriate software environments first before being able to combine the modules with the core framework.

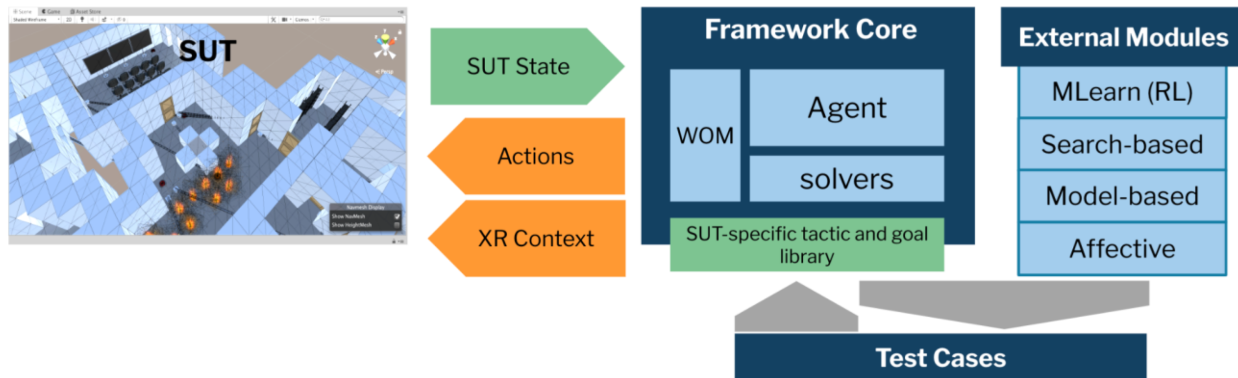


Fig. 2: *A top level architecture of the iv4xr Framework. The Framework Core refers to its agent-based programming and testing core. External modules refer to testing tools that are part of the Framework but outside the Core, though they use the Core.*

In this Report we will focus on the Framework Core. The external modules will be explained in the Reports of their respective Work Packages (e.g., explorative and search-based agents are explained in **D3.5** of WP3 and affective (user experience) testing is described in **D4.4** of WP4). This Report will also focus on a more conceptual presentation of the Framework Core. The GitHub-pages of respectively the Framework[1] and the Framework Core[2] provide instructions and further documentations on their usage. The Framework Core APIs documentation can be found here[3].

SECTION 2 - RELATED WORK

At the moment iv4xr is one of its kind in terms of providing automated testing for XR systems in the scale and spectrum that it can provide. In the current state of technology there are tools like Unity Test Framework[4] and GameDriver[5] that allow XR testing tasks to be scripted, hence they can be executed repeatedly by a computer rather than manually. These work fine for testing short scenarios but do not scale for long scenarios. The latter involve hundreds if not thousands of steps, which would be time consuming and fragile if we are to script them manually. Some tools like the aforementioned GameDriver and MAuto [TOK19] can record a scenario and replay it as a test. While this removes the need to script the test, record and replay tests are unfortunately

[1] https://github.com/iv4xr-project/iv4xr-framework
[2] https://github.com/iv4xr-project/aplib
[3] https://iv4xr-project.github.io/apidocs/aplib/javadocs/index.html
[4] https://unity.com
[5] https://www.gamedriver.io

also fragile. They break[6] easily when developers change the layout of the virtual world of the application that is under development or move the positions of the world's entities.

Automated testing (in the sense of generating the tests, rather than just providing automation in executing tests) in the XR domain has been mostly attempted in research setups. Much of recent work has been focused on using machine learning in the domain of game testing [ABS19,Hol+19,Mug+19,Zar19], e.g. where researchers explore the use of techniques such as Monte Carlo Search Tree (MTCS) and reinforcement learning (RL). Such ML techniques are used to train a procedural persona to play a game in a certain style (e.g an explorer persona or a killer persona) which is then used to automatically play the game for the purpose of testing it. Such an approach of testing is suitable for assessing the game balance against different styles of play. The approach is not specifically aimed at covering as much possible behavior as possible. Zheng et al investigated a combination of deep RL and an evolutionary algorithm to drive a more explorative agent towards maximizing coverage [Zhe+19]. Recent fascination in ML is understandable; but the approaches do have drawbacks:

- They require much training, which in turn requires executions on the system under test (SUT) for generating data. Unlike testing a library, executing on an XR system takes much more time. The overall computational requirement might be too expensive for small companies to afford.
- The resulting trained model might not be accurate, which may lead to false positives for developers to investigate.
- If the developers change the world layout, the trained model might not work anymore, and need to be re-trained, which is expensive.
- The aforementioned approaches are used to train an agent to accomplish a single goal (e.g. to reach the game's end state, or to maximize coverage). The setup is not really meant for automating multiple scenarios; for these, each scenario will require separate training, which further multiplies the cost.

More traditional approaches to automated testing are search based testing (SBT) [Min04], e.g. using evolutionary algorithms, and model based testing (MBT) [Net+07,IFT+15]. It is unclear whether "searching" would be more efficient than "learning". If done with little guidance (which usually is the case) SBT is also computationally intensive and suffers essentially the same drawbacks as mentioned above. On the other hand, MBT is fast and is much less energy hungry. However, we need to have a (behavioral) model of the SUT for it to work, which is quite costly to craft. In contrast, iv4xr relies on agent programming. We program tactics and strategies. They are composable to build various test scenarios. There is no need for training. The resulting test agents are precise, as their behavior is not controlled by some learned approximated functions. They are also robust with respect to various development time changes (that is, tests by iv4xr agents do not easily break) [Shi+21]. Iv4xr can also do MBT, if a model is provided (elaborated in Report **D3.5**). The MBT exploits agents as executors. This allows the models to be formulated at a higher

[6] By "breaking" a test we mean that the test fails, but not because of a bug. Rather it fails because the way it interacts with the system under test is no longer valid. E.g., it tries to interact with an entity that has been removed on purpose as part of the new logic of the new version of the system.

level, and hence simpler and more feasible to create. In a recent study we have also shown that iv4xr agents can be extended to construct a behavioral model as it explores the SUT [Shi+22].

Table 1 below gives an overview of some selected features of available XR testing technology and automated testing techniques, and how they compare to iv4xr.

	Various testing technology and automated testing techniques that have been tried on XR systems (mostly on computer games)	test scripting	record and replay	automated scenario testing	automated play testing	explorative testing	model based testing	affective testing
technology	Unity Testing Framework	✓	✗	✗	✗	✗	✗	✗
	GameDriver	✓	✓	✗	✗	✗	✗	✗
	MAuto	✗	✓	✗	✗	✗	✗	✗
research	ML/RL based approaches [ABS19,Hol+19,Mug+19,Zar19]	✗	✗	✗	✓	✗	✗	✗
	Search based approach [Zhe+19]	✗	✗	✗	✓	✗	✗	✗
	MBT [IFT+15]	✗	✗	✓	✓	✗	✓	✗
	iv4xr	✓	✗	✓	✓	✓	✓	✓

Table 1: *An overview of some selected features of available XR testing technology and automated testing techniques. The feature play testing refers to a start-to-end testing. Explorative testing refers to the testing of the SUT by interacting with it without any scripted scenario in order to reveal behavior left unexposed by scenario-based testing; this is also discussed in Report D3.5. Affective testing refers to the testing of user experience. This is discussed in Report D4.4.*

As for affective testing, there is still no publicly available solution that companies can use on their products, although there exist some initial research efforts on the problem from the side of academia [Mel+21, Fer+21, Ans+21, Mak+21].

SECTION 3 - AGENTS

This section explains how an iv4xr agent executes and how it interacts with its environment (the SUT). We also explain its generic representation of the world state and its memory feature. And finally, we also explain its support for automated world navigation and exploration.

Figure 3 below shows the top level architecture of an iv4xr test agent. Since the XR system under test (SUT) is not necessarily written in Java, we will conservatively assume that iv4xr test agents are deployed as programs that run outside the SUT runtime environment. To control the SUT the developers need to provide an interface. In Figure 3 this interface is called the *Environment*. It

should enable a test agent to control and observe the SUT the way its user can, e.g., to control it by simulating keyboard input. The agent has two more components: a *goal structure* and a *state*. A 'goal structure' is used to formulate a test scenario. It consists of one or more goals that form the scenario, along with 'tactics' describing how to achieve these goals. A state is just an object representing the agent's state (also called 'belief' in the BDI terminology). Importantly, it contains a data structure called *WorldModel* (**WOM**) that the agent uses to keep track of the state of the SUT's virtual world.

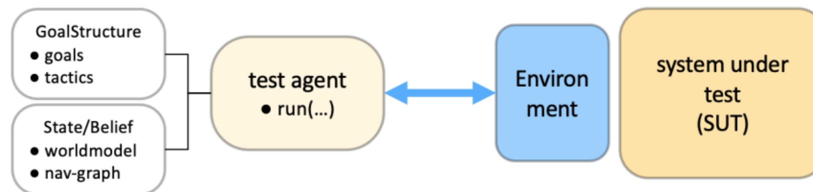


Fig. 3: *A top-level architecture of an iv4xr test agent. If needed, multiple agents can be deployed to control the SUT.*

The Environment. The Environment is assumed to provide a set of methods implementing primitive actions that a test agent can do on the SUT, e.g., to simulate a user's interaction with a nearby object in the SUT's virtual world, or to simulate a walk in the virtual world for some unit of distance. One of these methods should be *observe*() that returns an observation on the SUT state, for the rest the developers can decide which primitive actions the *Environment* offers and how much information *observe*() reveals. Making all objects fully observable would make testing easier, but this might be computationally excessive as the SUT might then need to keep sending the states of thousands of objects to the agent. To be realistic, we might also want to restrict the observation to reflect what an actual user can see.

```
1  TestAgent α       = new TestAgent("player-1") ;
2  GoalStructure testingTask = ...
3  LTL[] assertions = ...
4
5  α. attachState(new MyState())
6    . attachEnvironment(new MyEnvironment())
7    . setGoal(testingTask)
8    . addLTL(assertions) ;
9
10 while (testingTask.status().inProgress() α.update() ;
11 assertTrue(testingTask.getStatus().success() && α.evaluateLTLs()) ;
```

Fig. 4: *An example of setting up a scenario test in iv4xr.*

Setting up a test. Figure 4 shows a template code to set up a test using an iv4xr agent. Usually, a single test like this is used to test a single scenario. The code is in Java. In line 1 we create a test agent. The intended scenario to test is to be formulated as a goal structure, to be plugged in at line 2. Line 7 gives the goal structure to the agent. Before we can run the agent, it needs a state and an interface to the SUT; lines 5 and 6 create these and attach them to the agent. Line 3 is to be completed, to specify the *Linear Temporal Logic* (LTL) [BK08] assertions to be checked

during the play. Then, line 10 runs the agent, which would also check the LTL assertions as it goes. The last line checks that the specified scenario is successfully completed and that all LTL assertions were satisfied.

Agents' execution model. As mentioned, iv4xr implements BDI agents [Her+17]. Given a goal, a BDI agent runs an execution loop. Each iteration is also called a *deliberation cycle*: the agent deliberates which action to execute and executes it. Such a loop is highly reactive. When the next cycle's observation shows a change in the SUT state, the agent can immediately respond to it. In fact, many XR systems such as games and simulators are implemented using a similar loop. Each iteration is called a frame update. For example, a 3D simulator may visually appear to run smoothly, but it is actually a discrete system that performs one frame update at a time, but at a high speed of e.g. 100 frame updates every second. At every frame update the simulator basically iterates over every object in its world to update its state and visualization. As such, it is a very dynamic system (its state can change at every frame update), and having a test program that runs in a similar way is essential to enable it to keep up with the dynamics.

Algorithm 1 Agent's working loop; α denotes the agent. G is the goal structure to achieve.

```
1:  α.G ← G
2:  α.goal ← determine a goal to pursue                      ▷ α.goal ∈ α.G
3:  while α.G is not achieved or failed do
4:      α.update()
5:      α.sleep()                                            ▷ for some Δ ≥ 0 time
6:
7:  procedure α.update( )
8:      o ← env.observe()
9:      incorporate o to α.belief
10:     T ← α.goal.tactic
11:     proposal ← execute T
12:     if T aborted α.goal then
13:         α.goal.status ← Failed
14:     else
15:         if proposal ≠ ⊥ ∧ α.goal(proposal) = true then
16:             α.goal.status ← Achieved
17:     if α.goal.status ∈ {Achieved, Failed} then
18:         if α.G is also achieved or failed then
19:             return                                       ▷ we are done with G
20:         α.goal ← determine a next goal to pursue         ▷ α.goal ∈ α.G
```

Fig. 5: *An agent's execution loop.*

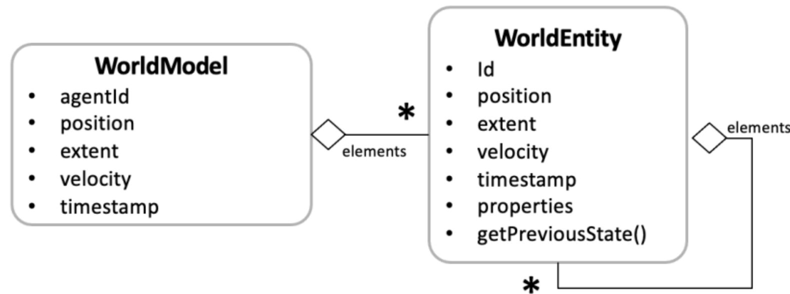
Programming a BDI agent comes down to programming its deliberation logic. In iv4xr, this logic is called *tactic*. Tactics are good for programming how to solve a simple goal. To formulate a complex test scenario, we use a goal structure, which essentially is a high level plan of how some main goal is to be achieved through subgoals. To support tactics and goal structures, iv4xr agents have an elaborate execution loop, shown in Algorithm 1 in Figure 5 above. Here, the agent is α. It gets a non-empty goal structure *G* —for now we can think of it simply as a set of goals. Each goal $g \in G$ should be accompanied with a *tactic*, denoted by *g.tactic*, used for achieving/solving *g*. Although *G* may contain multiple goals, an iv4xr agent works on one goal at a time. So, first it

chooses a goal to work on (line 2); how the choice is made will be discussed later. Then the agent runs its deliberation cycles in lines 3-5. The loop runs until the whole *G* is confirmed to be achieved or failed. In terms of testing, if achieving *G* is considered as an expected result, then failing it would count as failing the test. Note that when we run a test scenario, we may also add various assertions (as we did in the example in Figure 4) that are to be checked during the play, so failing *G* is not the only error that a test can discover. We will discuss assertions in Section 5.

Each deliberation cycle is done by the agent's *update*() method, which is shown in lines 7-20. In line 8 the agent calls the Environment's *observe*(), it then incorporates the obtained observation into its state. Then in line 11 it executes the current goal's tactic. This may change the SUT state. It also returns a 'proposal', which if accepted by the current goal, the goal is then achieved (line 16). After possibly many cycles of executing the tactic, we may come to a point where the current goal is either achieved or failed (line 17). The goal is failed when the tactic explicitly calls an 'abort' primitive (e.g. because it no longer believes the goal is attainable). Line 18 first checks whether this means that the entire *G* is achieved/failed. If so, the whole execution loop is completed. Otherwise in line 20 the agent selects a new current goal and then proceeds with its deliberation cycles.

Observation, WOM, and agent's memory. As with any program state, an agent state *S* may contain various variables. But importantly, it contains an object called *WorldModel* (also called 'WOM'), accessible through S.worldmodel. A *WOM* provides a generic representation of the state of a virtual world. It contains basic properties of the agent that owns it, e.g. its id and position in the world, and a set of *WorldEntities* accessible via *wom.elements*. A *WorldEntity e* represents an object *u* in a virtual world. Its notable fields are *e.id*, *e.position* and *e.properties* containing a set of name-value pairs representing the properties/state of *u*. Recall that at the start of every update cycle the agent samples a new observation on the SUT (line 8 in Algorithm 1). This observation o comes in the form of a WOM as well, representing the SUT state at that moment. Observations are 'aggregated' into *S.worldmodel* (line 9 in Algorithm 1). If for example in *o* contains a WorldEntity *v* that the agent has not seen before, *v* will be added to *S.worldmodel.elements*. It will be kept there ("memorized"), even if after some time the agent no longer observes it. The information will be updated if a fresh observation *o'* is received, that contains new information about *v*. *S.worldmodel* contains thus information on both the part of the SUT state that the agent currently sees as well as memorized properties of objects outside its current observation range. The latter information might indeed be partially or completely obsolete, but it might still be useful (it often is). It is up to the agent if it wants to use the information. The information is timestamped, so the agent knows how old it is.

The UML class diagram below shows the structure of *WorldModel* and *WorldEntity*.



Navigation. An important part in XR test automation is automated navigation/travel over the world terrain. Since the walkable area of a virtual world is often an infinite domain (consisting of infinite number of visitable points), a common way to calculate navigation routes is by representing it as a *navigation graph* (or simply 'nav-graph') $\theta = \langle F,V,E \rangle$, where F is a finite set of obstacle-free and non-overlapping small areas called 'faces' that cover the entire walkable area (e.g. small squares or triangles can be used as faces). Note that if two faces $a,b \in F$ are adjacent, then the player can travel from any point in a to any point in b. V is the set of nodes in the graphs, consisting of locations, such that for every $a \in F$, it is represented by one node $p \in a$. E is the set of edges connecting the nodes, such that two nodes are connected when the faces that contain them are adjacent. We assume the agent state S also holds a nav-graph. Such a graph can be exported by the SUT itself, or else iv4xr provides a utility to construct it on the fly. Iv4xr Framework Core provides an implementation of the path finding algorithm A* [Pra+20] to calculate a path between two nodes. An agent can then be programmed to follow the path. We also assume that *observe*() also reveals which nodes in the nav-graph are currently visible to the agent. So, we can mark which nodes in the nav-graph that the agent has seen so far. We can thus calculate the so-called *frontier nodes*: marked nodes with at least one unmarked neighbor. Sending the agent to visit a frontier node will cause it to discover more nodes, which can be used to drive exploration [kwe97,Pra+20].

To summarize, the following two functionalities are provided by the Framework Core: if p,q are positions in the SUT's virtual world, *findpath*(θ,p,q) returns a sequence σ of nodes in V specifying a traversable path to go from p to q, if there is such a path, otherwise it returns $\perp$. The path σ starts with a p' in the same face as p, and ends with a q' in the same face as q. *frontiers*(θ) returns the set of current frontier nodes of θ .

SECTION 4 - TACTICS AND GOALS

To help explaining the concepts below, we will use the 3D game shown in Figure 1 as a *running example.* The game is called *LabRecruits*. There are objects in the game that the player's avatar can interact with, such as switches. Access to areas/rooms can be guarded by doors, whose state can be toggled by interacting with a matching switch. There can be many-to-many connections between switches and doors, which makes getting access to an area non-trivial. There are also hazards such as fires and monsters.

The Environment that interfaces with this game is assumed to include the following methods:

- *observe*() : returning an observation from the perspective of the player's avatar.

- *moveTo*(*v*): move the avatar to a location *v* provided this location is close to its current location (reachable in one frame-update) and can be reached with a straight-line walk.
- *interact*(e): make the avatar interact with a game object *e*, provided *e* is close enough to the avatar.

The agent can access its Environment through its state, as in *S.env*().

ACTION

Recall that at every deliberation cycle an agent executes the *tactic* of its current goal (and the cycle is repeated until the goal is achieved or aborted). A tactic is made of 'primitive actions'; these are the methods provided by the Environment to provide control over the SUT (this was discussed in Section 3). These primitives are meant to be simple and usually correspond to basic interactions that a user can do, e.g., to walk for a small distance within a virtual world, or to interact with an object located in a close proximity of the user's avatar in the SUT's virtual world. To provide logic on when an action can be executed, a guard can be attached to it. The syntax for defining an action is shown below:

$$\textbf{var}\ \alpha\ =\ \textbf{action}(\text{"}id\text{"})\ .\ \underbrace{\textbf{do}_2(f)}_{\text{behavior}}\ .\ \underbrace{\textbf{on}_(q)}_{\text{guard}}$$

This constructs an action, and binds it to the variable α. The *f* part forms the body of the action, this is where we would call a method from the Environment. The *q* is the guard attached to the action. An action is only *enabled* (eligible for execution) when its guard is true on the current agent state. While it is common to specify the guard using a predicate (a function that returns true/false), the pair (*q*,*f*) actually forms a monadic bind [Wad92]. The *q*-component can be a *query* function that inspects the current agent state and returns *v*. The action α is enabled only if the returned *v* is not null, and additionally the behavior of *f* can be made to depend on the returned *v*. For instance, such a query can be used to *find* an object *e* in the agent's WOM which is close enough to the player's avatar. Then *f* can be an action that moves the avatar closer to *e*. Note that this construct is more powerful than conventional true/false guards.

As an example, consider the LabRecruits game introduced at the beginning Section 4. Below we show an action *navigateTo*(id) that uses the aforementioned pathfinder to guide the agent to the location of an in-game object *e* with the given id. Notice how the guard-part of the action is actually a query to find a path to *e*.

```
navigateTo(id) = action()
  .do2 (S → path → S.env().moveTo(path.get(0)))
  .on_(S → { e = S.worldmodel().elements.get(id) ;
             agentposition = S.worldmodel().position ;
             return pathfinder(S.worldnavigation,agentposition,e.position) ; })
```

Note that the action will not reach *e* immediately, e.g., if *e* is far from the current agent location. It is meant to be executed in an agent loop as in Figure 5, where it will eventually reach *e*[7].

As another example, below is an action *explore*(), to guide the agent to the closest and reachable frontier node, from where it could see/discover a part of the world it has not seen before. The action has the same behavior part as *navigateTo*, but uses a different query.

[7] As another note is that invoking the pathfinder at every deliberation cycle is obviously wasteful. It can be made efficient e.g., using memorization. In favor of readability, we do not show such optimization.

```
explore() = action()
   .do2 (S → path → S.env().moveTo (path.get(0)))
   .on_(S → // use frontier(S.worldnavigation,agentposition) to check if there is
              // a reachable frontier node; if so return a path to the closest of such a node )
```

TACTIC

An action can be lifted to a tactic. A more powerful tactic can be obtained by combining multiple actions. In principle a tactic is a set of actions. It is enabled when one of its actions is enabled. When the agent executes its current tactic, one of the tactic's enabled actions will be executed. If there is none, the agent will just do a skip-step, hoping that in the next deliberation cycle, the SUT state changes so that at least one action becomes enabled. More precisely, a tactic is a *hierarchical* composition of actions. Tactic *combinators* are used to combine actions. An example is shown below:

```
1  var navigateToTac(e) = FIRSTof(
2     unstuckTac(),
3     navigateTo(e).lift() ,
4     explore().lift() ,
5     ABORT)
```

Fig. 6: *an example of a tactic, constructed from a number of actions and sub-tactics combined with the **FIRSTof** combinator.*

Above, the tactic *navigateToTac*(*e*) uses the previously introduced *navigateTo*() action, but it also uses other actions/sub-tactics, combined using the **FIRSTof** combinator. This combinator chooses the first enabled sub-tactic to be executed. This is useful if the agent has not actually seen the object *e* (so it does not know its position either) the action *navigateTo*(e) in line 3 will not be enabled, in which case the tactic above would fall back to *explore*() in line 4 that will drive the agent to explore the world to discover previously unseen parts of it. If at some point it sees e, the action *navigateTo(e)* will be enabled and the agent can switch to it. If *explore*() runs out of frontier-node (in other words, the agent has explored what it can) and *e* is still not found, the tactic exits through **ABORT** in line 5, which would abort the agent's current goal.

There is one other sub-tactic with a higher priority in *navigateToTac*(), namely the one in line 2 of Figure 6. The tactic is enabled when the agent becomes stuck (its position remains the same in the last k-cycles, e.g. because it is halted by a small sticking obstacle). The tactic *unstuckTac*() would then try to unstuck the agent e.g. by moving it a small distance to the left or right. Unstucking can be quite complicated, but the important thing to note here is that we often need this kind of priority-based logic when controlling an agent, and the **FIRTSof** combinator gives a nice way for expressing such a control.

An important thing to note is that while it is possible to embed all the actions' logic in their guards, combinators like **FIRSTof** provide a cleaner way to program common controls like priority-based control. Available combinators are shown below.

Tactic ::= *GuardedAction*.**lift**()
| **ANYof**(*Tactic*, ..., *Tactic*)
| **FIRSTof**(*Tactic*, ...,*Tactic*)
| **SEQ**(*Tactic*, ..., *Tactic*)

Fig. 7: *Tactic combinators for iv4xr agents.* ***ANYof*** *randomly chooses an enabled sub-tactic.* ***SEQ*** *executes the sub-tactic in the specified sequence.*

Goal and Goal Structure

The syntax to define a goal for an agent is shown below:

$$Goal \quad ::= \quad \mathbf{goal}(name).\mathbf{toSolve}(\overbrace{Predicate}^{State \to Bool})\,.\mathbf{withTactic}\,(Tactic)$$

The predicate part is used to specify desired states to be. The tactic specifies how the agent can execute actions to achieve the goal; that is, to reach a state satisfying the goal-predicate. Two examples for LabRecruits are shown below.

```
1 Goal closeBy(id) { return
2     goal()
3     .toSolve(S → { e = S.worldmodel().elements.get(id) ;
4                    agentposition = S.worldmodel().position ;
5                    return (e != null  && dist(agentposition,e.position) < ε) ; } )
6     .withTactic(navigateToTac(id))
7
8 Goal interacted(id) { return
9     goal()
10     .toSolve(S → true)
11     .withTactic( // interact with the entity identified with id, if the avatar is close enough to it )
```

Fig. 8: *Two elementary goals for the game LabRecruits. The goal closeBy(e), aims to get the agent to some location close to e. We use the previously introduced tactic navigateToTac(e) to solve this goal (line 6). The goal interacted(e), line 8, aims to get the object e interacted by the agent. The tactic is not shown, but it assumes the agent to be close to e. Elementary goals can be composed to make a more complex one, e.g.* ***SEQ****(closeBy(e), interacted(e)) will guide the agent to get close to e, and then interact with it.*

Tactics are usually good for 'solving' (achieving) simple goals. To handle a harder goal, we can specify a set of subgoals, each is easy enough for a tactic to solve. For example imagine in our example game we have to test some feature of some game object *F* located in some specific room. Let *isInteracted*(*F*) be the goal representing the agent is at *F* and manages to interact with it (and tests its feature as it does so). To achieve this the agent will first need to reach the room where *F* is. To access this room a door *D* needs to be opened first. The door can be closed, in which case the agent first needs to find a specific in-game switch *B* that opens it. Obviously, the previous tactic *navigateToTac*(*F*) will not be able to solve *isInteracted*(*F*) on its own, as it has no knowledge that there is an additional switch-logic that it needs to engage before it can get to *F*.

A *goal structure* is a way to express a goal in terms of a composition of subgoals. Goal-combinators are used to compose the subgoals. The simplest combinator is lift(), to lift a simple goal to become a goal structure. Another example is the combinator **SEQ**. If G_1, ... , G_n are goals/goal-structures, G = **SEQ**(G_1, ..., G_n) is a goal structure that seeks to solve all its subgoals

in the given order. When this happens, *G* itself is considered as solved. If one of the subgoals fails, *G* fails. For example, the above described test scenario can now be formulated as the following goal structure:

```
1 GoalStructure test_F = SEQ(
2     SEQ(closeBy(b0),                  // b0 is a switch
3         interacted(b0),   // toggle the switch, this should open door d0
4         closeBy(d0),
5         assertTrue_(...,S → // check that d0 is open ),
6         closeBy(F),
7         interacted(F)),
8     assertTrue_(...,S → // check the state of F) )
```

Fig. 9: *An example of a goal structure. This goal structure will first guide the agent to get to the switch b0 to toggle it, then to door d0, and check whether it is open. Then it drives the agent to get to the entity F, interact with it, and then check its state. Assertions (blue) to check are added with assertTrue_(ψ) constructs. Also notice that the goal-structure is structured in two parts. The inner SEQ is meant to establish a state where F has been interacted, then the second part (line 8) is an assertion meant to check the state of F after it has been interacted.*

The set of basic goal combinators is shown below:

$$
\begin{aligned}
GoalStructure \;\; ::= \;\; & Goal.\mathbf{lift}() \\
| \;\; & \text{SUCCESS} \\
| \;\; & \text{FAIL} \\
| \;\; & \text{SEQ}(GoalStructure, \ldots, GoalStructure) \\
| \;\; & \text{FIRSTof}(GoalStructure, \ldots, GoalStructure) \\
| \;\; & \text{REPEAT}(GoalStructure)
\end{aligned}
$$

SUCCESS is a goal that always succeeds, whereas **FAIL** always fails. **FIRSTof** executes the subgoals in sequence. It succeeds at the first subgoal that succeeds (the rest of the subgoals will then not be executed), and else fails when all subgoals fail. **REPEAT**(*G*) repeatedly tries G until it succeeds.

Section 5 - Specifying Assertions

In testing "*assertion*" is a term used to refer to an expression used to check correctness. It can be the correctness of the current state, or the correctness of a series of states. Bear in mind that we cannot always assume that test agents have direct access to the SUT state. This depends on how the Environment component is implemented. So, in our setup assertions will be checked on the agents' states rather than on the actual SUT states. However, recall that an agent state incorporates a WOM, which in turns incorporates the agent's most recent observation as well as memorized states of various objects the agent observed in the past. The WOM contains thus the best information the agent has about the SUT state.

ASSERTING REACHABILITY

In the previous section we have seen that a test scenario can be formulated as a goal or a goal structure *G*. If the scenario is expected to be achievable, then the status of *G* at the end of the agent's run can be used as an assertion, namely that *G.status*().*success*() should be true. We did this in the example in Figure 4. However, there are more types of assertions we can express in iv4xr. But let us first introduce the concept of "*testing task*" that generalizes the concept of test scenario.

The first type of testing task is a task to check the reachability of some state predicate $\boldsymbol{\phi}$. More precisely:

Def. 1: a state predicate $\boldsymbol{\phi}$ is *reachable* if there exists a state s which is reachable from the agent's initial state s_0 and such that $s \vDash \boldsymbol{\phi}$.

Def. 2: a state t is *reachable* from another state *s* if there exists a sequence of actions $\boldsymbol{\sigma}$, such that when this $\boldsymbol{\sigma}$ is executed on the state *s*, it results in the state *t*.

A testing task to check the reachability of a $\boldsymbol{\phi}$ can be directly formulated as a goal, with $\boldsymbol{\phi}$ as the predicate to solve, and assert, at the end of the agent's run, that the goal is achieved/solved. If it is hard to solve $\boldsymbol{\phi}$ directly with a tactic, subgoals can be introduced to provide more guidance as discussed at the end of Section 4.

ASSERTING INVARIANT

The second type of testing tasks if a task to check $\boldsymbol{\phi} \Longrightarrow \psi$, where both $\boldsymbol{\phi}$ and ψ are state predicates. The implication is defined as follows:

Def. 3: $\boldsymbol{\phi} \Longrightarrow \psi$ means that for all *reachable* state *s* such that $s \vDash \boldsymbol{\phi}$, we also have $s \vDash \psi$.

Such an implicative property is a generalization of the concept of *invariant* as in Enrst et. al's Daikon [Ern+07], who define an invariant as a state predicate that always holds on a certain control location in a program. Above, we replace "control location" with a predicate $\boldsymbol{\phi}$ that characterizes SUT states of a certain property. The ψ captures the invariant that is expected to hold whenever the SUT is in the state $\boldsymbol{\phi}$. Note that the definition is constrained that ψ is only required to hold on the reachable part of $\boldsymbol{\phi}$, rather than on the entire $\boldsymbol{\phi}$.

In a testing setup we cannot really check the implication on all states satisfying $s \vDash \boldsymbol{\phi}$. Instead, we try to find *one* such *s*, and then we check if ψ holds on it. This can be implemented as a goal structure *G* = **SEQ**($G_{\boldsymbol{\phi}}$,assertTrue_(ψ)) where $G_{\boldsymbol{\phi}}$ is a goal structure that seeks to establish a state satisfying $\boldsymbol{\phi}$. An example of this was shown in Figure 9. If checking ψ on more witnesses is desired, we can for example split $\boldsymbol{\phi}$ into disjunct $\boldsymbol{\phi}_1 \vee \ldots \vee \boldsymbol{\phi}_n$, and then verify each $\boldsymbol{\phi}_k \Longrightarrow \psi$ individually.

Checking ψ is usually easy, it is finding a proper s (one such that $s \vDash \boldsymbol{\phi}$) that is actually *much* harder as it involves finding a sequence of actions $\boldsymbol{\sigma}$ that leads to such an s. This sequence $\boldsymbol{\sigma}$ is often *very long*, and finding it requires insight on the SUT's logic. In the example in Figure 9 we showed how this can be done by providing subgoals, each is intended to guide the agent in finding

some key segment of σ. Further automation by employing online search algorithms[8] that enables the test agent to find some, and sometimes all, these subgoals on its own. These algorithms were presented in [Shi+21,Shi+22]. Employing an online search algorithm can however be quite costly in terms of computation time. In [Shi+22] we have shown that the agent can also be extended so that it also builds a model of the SUT's logic while it is performing the search. Using such a model, next time we want to execute the same testing task, the search can be done fully or partially on the model, which can be done much faster.

ASSERTING DYNAMIC PROPERTY

The third, and last, form of testing tasks that an iv4xr agent can perform is a task to check the implication $\phi \Longrightarrow ltl(\psi)$, where ϕ is a state predicate and ψ is an Linear Temporal Logic (LTL) formula. An LTL formula is a predicate of over sequences of states, so it allows us to capture a requirement over a whole execution rather than just a requirement over some states. For example, for our Lab Recruits game example, we may want to check that for some ϕ the total amount of points collected by the agent during the execution should *never* exceed 100. Another example: the state where a door *d* is open, but a switch *b* is toggled off should never occur.

Formally, the implication is defined as follows:

Def. 3: $\phi \Longrightarrow ltl(\psi)$ means that for all "executions" π that ends in an s such that $s \models \phi$, we also have $\pi \models \psi$.

An execution here means a sequence π of states starting at the agent's initial state s_0 (so $\pi_0 = s_0$, and such that for any two consecutive states π_k and π_{k+1} there is an action *a*, which if it is executed on π_k it would result in the state π_{k+1}.

As before, in a testing setup we cannot check the implication on all possible executions, but we can check it for one sample execution. This can be implemented simply by offering a goal structure G_ϕ that seeks to establish the state predicate ϕ. One or more LTL formulas can be given to the agent, which it then checks while executing G_ϕ. This scheme checks the implication on one witness execution. If more witnesses are desired, as before we can split ϕ into disjunct $\phi_1 \vee \ldots \vee \phi_n$, and then verify each $\phi_k \Longrightarrow ltl(\psi)$ individually.

LTL is a well known formalism for expressing sequence predicates [BK08]. The syntax for writing LTL formulas in iv4xr is shown below, where *p* is a state predicate as an atom, φ is an LTL formula, and *F* is either a state predicate or an LTL formula.

F ::= *p* | φ	
φ ::= **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**($\varphi_0, \ldots, \varphi_{n-1}$)	-- conjunction ∧
| **ltlOr**($\varphi_0, .., \varphi_{n-1}$)	-- disjunction ∨
| **ltlNot**(φ)	-- negation ¬
| φ.**implies**(*F*)	-- implication → operator
| φ.**until**(*F*)	-- the **U** operator

[8] An "online" search algorithm executes directly on the SUT. In contrast, an offline algorithm performs the search on a model. An offline approach is much faster, but obviously we can only do it if we have a model.

The semantics of LTL formulas are usually defined over infinite sequences [BK08]. However, since executions in tests are of finite length, we need to use a variant that defines their semantic over finite sequences. We use the definition as in [Pra18]. In particular, this definition judges $\boldsymbol{\pi} \vDash \varphi$ based only on the information in $\boldsymbol{\pi}$. If φ is a future operator, then the promised situation must be observed in $\boldsymbol{\pi}$. Conversely, if φ is a safety property, the definition does not consider possible violations if $\boldsymbol{\pi}$ would be extended. For example $\boldsymbol{\pi} \vDash$ **eventually**(p) holds if p holds on some state $\boldsymbol{\pi}_i$, whereas $\boldsymbol{\pi} \vDash$ **always**(p) is considered to hold if all states in $\boldsymbol{\pi}$ satisfies p (even if some future extension of $\boldsymbol{\pi}$ might violate p). For evaluating the formulas we implement a labeling algorithm as in [Pra18][9] that allows linear time (with respect to the sequence length) evaluation.

As examples, below are some LTL specifications we could impose on the execution generated by the test in Figure 9. The first one f_1 states that eventually the agent should accumulate 11 points (a player gets points for e.g. getting close to a switch and toggling it for the first time). The second one f_2 states that it should never be the case that door d_0 is open while the switch b_0 is turned off.

```
var f1 = eventually(S → {
        agent = S.worldmodel().elements.get(agentId) ;
        point = agent.properties.get("point")
        return point >= 11 ; } )

var f2 = ltlNot(eventually(S → {
        d0 = S.worldmodel().elements.get("d0")
        b0 = S.worldmodel().elements.get("d0")
        return b0 != null && d0 != null && ! b0.properties.get("isOn") && d0.properties.get("isOpen") ; } )
```

These formulas can be attached to the agent using *agent.addLTL*(f_1,f_2,...). When the agent is run, the attached formulas will be checked.

Reporting

In Java a test is usually coded as a JUnit test class. So an iv4xr test code such as in the example in Figure 4 would be put in a test method inside a JUnit test class. Various assertions, as mentioned above, would then be implemented as JUnit assertions. A reachability assertion can be directly checked by a JUnit assertion. The results of the other two types of assertions are logged into a *DataCollector* object, after which can be inspected by JUnit assertions. More of this is detailed in the Framework Core manuals[10].

Additionally, a test agent can be configured to produce a trace file when it is run. Values from the agent state (e.g. the agent's position, properties of key objects, etc) can be recorded into the trace file. Collected traces can subsequently be subjected to post-analyses, e.g. visualization and

[9] But without the algebraic part. [Pra18] allows a mix of algebraic and LTL properties to be formulated. The algebraic part is more costly to check. In iv4xr we do not implement the algebraic part. This is future work. For the pure LTL part its labeling algorithm can evaluate LTL formulas in linear time with respect to the length of the input sequence.

[10] https://github.com/iv4xr-project/aplib/blob/master/docs/agentbasedtesting.md

"runtime verification"[11]. An example of visualization is shown below; it is a heatmap showing the maximum intensity of some chosen numeric property in the agent state as it traverses a world.

Instructions on how to produce traces and visualization can be found in the above mentioned manuals.

The Framework supports the use of LTL to do runtime verification [BY18] on the produced traces. This can be done with the same LTL support discussed in the previous subsection. Whereas previously we discussed the use of LTL formulas as assertions to check live test executions, for runtime verification we use LTL post-mortem on trace files. We can also use LTL formulas to query the traces, e.g. if we are interested in finding traces showing a certain behavior. The Framework also provides an outside-Core module called *LTL-pxevaluation*[12] with an extended version of LTL which includes e.g. predicates over areas and aggregation. With this extension we can express, for example, that while the agent is in area A, the value of some property x never exceeds some constant C_1, or that the average of x while in A never exceeds C_2.

The trace files are in the CSV format. So, beyond the above two kinds of post-processing provided by the Framework, they can be targeted by general purpose data science libraries like Pandas.

SECTION 6 - DYNAMIC GOALS

Essentially, a goal structure is used to impose an order in which its goals are executed. This order can be seen as a plan. The goal-combinators (see Section 4) have one shortcoming though: the induced plan is rather static. For example, consider **FIRSTof**(G_1,G_2). In some SUT state G_1 might succeed, and hence G_2 will not be tried, but in a different state G_1 might fail, and hence both G_1 and G_2 are tried. While there is some dynamics in this, note that in both cases the order in which the subgoals are tried is fixed: first G_1 then possibly G_2. This fixation limits the way the agent can respond to emerging situations.

A dynamic goal is a goal structure whose structure can change at the runtime, and hence the plan it induces can also change dynamically. The following *actions* are the primitives to do such runtime change:

[11] Contrary to its name, "*runtime verification*" does not refer to verification while a program is running. Rather, it is verification based on extracting information from a running system [BY18]. The approach has been researched, with various approaches have been developed.

[12] https://github.com/iv4xr-project/ltl-pxevaluation

$$\begin{array}{ll} \textbf{addBefore} & (\overbrace{goal\ function}^{State\rightarrow GoalStructure\ \text{or}\ State\rightarrow\alpha\rightarrow GoalStructure}) \\ \textbf{addAfter} & (goal\ function) \end{array}$$

Above, a 'goal function' is a function Γ that evaluates the agent current state and constructs a goal structure. Alternatively, it can additionally take a query result α as in the $\textbf{do}_2$ construct discussed in Section 4.

Suppose *g* is the agent's current goal. Executing *addBefore*(Γ) on the current state *S* will first construct a goal structure *H* = Γ(*S*), which will be inserted 'before' *g*, and then *g* itself is aborted. If g is a *k*-th subgoal in a **SEQ**(..*g*..) or **FIRSTof**(..*g*..), *H* will be inserted in the construct, just before *g* (so it will become the new *k*-th subgoal). If *g* is the root goal, or if it is a subgoal in **REPEAT**(*g*), we will treat it as a singleton **SEQ**(*g*) and then insert *H* accordingly. The combinator *addAfter* works similarly, but it inserts the new goal *H* after the current goal *g*. As any other action, *addBefore*(Γ) can also be guarded (with a predicate or a query). This allows the agent to decide at the run-time when it wants to deploy a new goal. The goal function Γ decides which goal is to be deployed. By default, goals added dynamically like this will be removed again after they are accomplished or aborted, so they cannot be executed again unless they are explicitly re-introduced.

An example for LabRecruits is shown below. Recall the *closeBy*(*e*) goal in Figure 8, used to drive the agent to a position close to the object *e*. The route to *e* can be long and hazardous. Hazard such as fire and enemies lower the player's health when they touch the player. The player can heal itself by touching a healing-pole, if it sees one (but this can only be done once per healing-pole). We can make the goal *closeBy*(*e*) smarter by introducing an action that can interrupt the current goal to first drive the agent to a healing-pole to heal up when its health becomes too low. This logic can be built with *addBefore*(), as shown below:

```
1  Action healUp() {return
2     addBefore(S → healingPole →
3                         SEQ(closeBy(healingPole.id), interacted(healingPole.id))
4     .on(S →
5              // check if (1) the agent health is low, and
6              // (2) there is an unused and reachable healing-pole in S.worldmode().
7              //      If one can be found, return it, else return null
8         ) ;  }
```

In the code above, lines 2-3 use a lambda-expression to define a goal-function. The goal-structure to generate is specified in line 3. Notice that the specific goal to do depends on the parameter *healingPole*, which is decided dynamically (at runtime).

We can now write a smarter *closeBy*(e), as shown below. It uses the original *closeBy*(*e*), but alters its tactic so that the action *healUp*() can fire first when its logic concludes that it needs healing and there is an unused healing-pole it can reach. When this happens, the goal structure changes from **REPEAT**(*closeBy*(*e*)) to **REPEAT**(**SEQ**(*H*,*closeBy*(*e*))) where *H* is the new goal to heal up the agent. The insertion of *H* also causes the goal *closeBy*() itself to be aborted. The **REPEAT** construct will now cause the **SEQ**(..) body to be tried again. If *H* succeeds, it is removed and we move on to *closeBy*(*e*). If *H* fails it is removed and **REPEAT** will retry *closeBy*(*e*). This goes on until the latter succeeds.

```
1 GoalStructure smarterCloseBy(id) { return
2       REPEAT(closeBy(e)
3               .withTactic(FIRSTof(healUp().lift(),  navigateToTac(e)))
5       ; }
```

Composing dynamic goals. Using the primitives *addBefore* and *addAfter* we can now define goal combinators that behave more dynamically than the ones we have in Section 4. For example we can now define **IF**(*q*, Γ,*H*). This constructs a goal structure that would first apply the query function *q* on the current agent state *S*. If this results in a non-null value *a*, it will continue with the goal Γ(*a*). Else, if *a* is null, it will continue with *H*. Figure 10 below shows how this combinator is defined using *addAfter*. The more traditional **IF**(*g,G,H*) where *g* is a predicate can be defined by encoding *g* as a query function. Note that **IF** cannot be defined using **FIRSTof**, because e.g. **FIRSTof**(*G,H*) always tries *G*, rather than choosing between *G* and *H*.

Recall that we also have a **REPEAT**(*G*) construct; this was introduced in Section 4. This construct repeatedly attempts the goal *G* until it succeeds. Using **IF** we can now also have guarded **REPEAT** and **WHILE** constructs; see Figure 10.

```
GoalStructure IF(q,Γ,H) {
    Goal g = toSolve(x → true)
            . withTactic(addAfter(S → {
                  a = q.apply(S) ;
                  return a!=null ? Γ.apply(a) : H ; }).lift()) ;
    return FIRSTof(g.lift()) ; }

GoalStructure IF(g,G,H) { return IF(S → g.test(s) ? true : null,   b → G,  H) ; }
GoalStructure REPEAT(G,g) { return REPEAT(  SEQ(FIRSTof(G,SUCCESS), IF(g,SUCCESS,FAIL)))
GoalStructure WHILE(g,G)   { return IF(g, REPEAT(G, not(g)), SUCCESS) ; }
```

Fig. 10: *Combinators to construct dynamic goals.*

SECTION 7 - DEPLOYMENT ARCHITECTURE

As a framework iv4xr is not something we can use out of the box. We first need to construct several components to connect it to a given SUT, and to provide some basic automation, which later at the higher level can be combined to deliver more powerful automation. The figure below shows the typical architecture for facilitating iv4xr use. Some of these components are SUT-specific, so it is not possible to provide them generically. Building these will require some effort; but it is a one-off investment, after which the built infrastructure can be reused over and over to do automated testing.

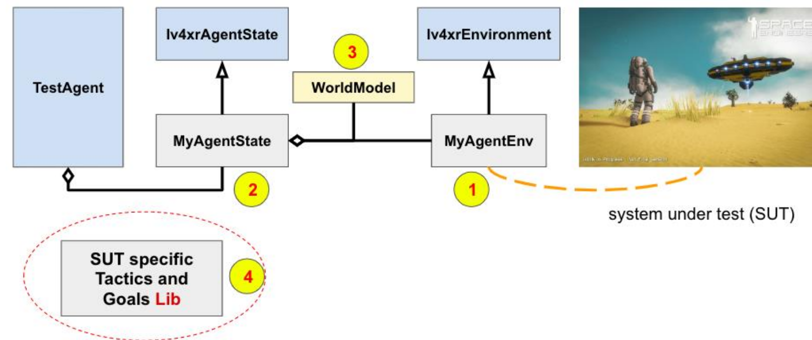


Fig. 11: *A typical architecture to integrate iv4xr into the testing architecture of an SUT.*

The test agent comes from iv4xr, or at least we need iv4xr Framework Core. Along with a test agent, the following components are needed:

1. An implementation of the class *Iv4xrEnvironment*, responsible for handling the interaction with the system under test (SUT), such as sending a command to the SUT and to obtain observation of the SUT state.
2. A direct instance or an implementation of the class *Iv4xrAgentState* to hold the agent's state. Among other things, this state will hold a WorldModel as a generic representation of the SUT's gamestate.
3. A *WorldModel* also contains one or more *WorldEntity*; each represents an object in the SUT. WorldModel and WorldEntity are generic representations, regardless of the SUT. We will thus need to build a translator that translates actual SUT objects and state to WorldEntity and WorldModel.
4. To do something smart, such as automatically navigating to a destination, our agent will need a bunch of tactics and goals. These are game-specific, so we will also need to construct a library of goals and tactics. Iv4xr does provide some support e.g. by providing pathfinding and exploration algorithms, in addition to a whole range of tactic and goal combinators.

More instructions and examples on how to set these up can be found in the Github page of the Framework[13].

Section 8 - Experience

Below we show some selected results from our studies. The first one is shown in Figure 12., taken from [Pra+22], is a study of a controlled subject, which is a small computer game called MiniDungeon. Despite the size the game offers quite some challenges such as limited observability, non-determinism, enemies, and guarded areas (areas guarded by doors that need to be unlocked first before the user can access the areas). While it is easy to do unit testing on separate entities/objects of this game, it is much harder to manually script a lengthy test scenario. The results in the table below show that with automated agent-based testing we can *significantly* improve the overall test coverage (compare U_{cov} and all_{cov}), and also find bugs that were not found by unit testing.

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

	C	i	cc	U_{cov}	PT_{cov}	all_{cov}	U_{bug}	PT_{bug}
Entity	11	360	21	81%	97.8%	100%	2	
Maze	1	320	18	89.9%	95%	95%	1	
MiniDungeon	2	2282	197	14.2%	84.4%	84.5%		2
MDApp	1	1228	89	0%	92.3%	92.3%		
All	15	4790	325	18.8%	88.2%	88.4%	3	2

Fig. 12: *Comparing the code coverage of manually written unit tests on a small game called MiniDungeon (U_{cov}) and the code coverage of automatic agent-based tests (PT_{cov}). The combined coverage is denoted by all_{cov}.Each row refers to different types of components of the game. C is the number of classes that implement the component-type that the row represents, i is the total number of instructions, and cc is the total cyclomatic complexity of the component type. U_{bug} is the number of bugs discovered by unit testing, PT_{bug} is the number of bugs discovered by agent testing.*

The next table in Figure 13 shows the results of a study [Shi+21] on the robustness of agent-based tests. A test is robust if it does not break when the developers change the SUT in a way that does not influence the feature that is being tested. For example, such a change can be a change in the world's layout, or if the developer moves the location of some objects to new locations, but it can also be some changes in the logic of the SUT. If a test breaks, then manual effort has to be spent to fix it. So, robustness is a desired property. E.g. by exploiting pathfinding discusses discussed in Section 3, a test agent would be able to deal with changes in the world layout, for example, as long as they do not influence the reachability to the objects it want to test.

In the study two setups were used. The first setup uses tactics that exploit pathfinding, but the goal structures are static (roughly they look like the example in Figure 9). The second setup exploits dynamic goal structures mentioned in Section 6. A dynamic goal is used to look for an alternative interactable that could unlock a block if the default one fails (e.g. because the logic has been changed).

The results in Figure 13 show that the first setup is quite robust with respect to layout and position changes, but not against changes in the logic of the SUT. The second setup is robust against both types of changes.

Mutation Type	Level	$Test_{base}$ success (p-val)	$Test_{dynamic}$ success (p-val)
Location change	$Level_1$	50/50 (0.0003)	50/50 (0.0003)
	$Level_2$	49/50 (0.0029)	49/50 (0.0029)
	$Level_3$	48/50 (0.0142)	48/50 (0.0142)
Location and logic change	$Level_1$	-	50/50 (0.0003)
	$Level_2$	-	50/50 (0.0003)
	$Level_3$	-	50/50 (0.0003)

Fig. 13: *the robustness of agent-based tests against changes in the world layout and changes in the SUT logic. The experiment is done on three levels of the game Lab Recruits.Two setups were compared. The first on, $Test_{base}$, uses static goal structures as in the example in Figure 9. The second one, $Test_{dynamic}$, exploits dynamic goal structures mentioned in Section 6.*

SECTION 9 - CONCLUSION

We have presented the main concepts of iv4xr agent-based testing approach. Its agent-based Core allows test agents to be programmed with tactics. Test scenarios can be abstractly expressed as single goals, whereas more complex scenarios can be expressed with goal structures. Automated testing through agents can significantly improve the overall coverage and strength of testing, on top of existing manual testing. Agent-based tests are also robust. However, due to the unstandardized technology used in XR systems, iv4xr requires an interface and a SUT-specific library of basic tactics and goals to be developed as well. However this is a one-off investment, after which developers can continuously benefit from the automation provided by iv4xr.

REFERENCES

[Ana+13] Anand, S., Burke, E.K., Chen, T.Y., Clark, J., Cohen, M.B., Grieskamp, W., Harman, M., Harrold, M.J., Mcminn, P., Bertolino, A., et al.: An orchestrated survey of methodologies for automated software test case generation. Journal of Systems and Software 86(8) (2013).

[ABS19] Sinan Ariyurek, Aysu Betin-Can, and Elif Surer. 2019. Automated Video Game Testing Using Synthetic and Human-Like Agents. IEEE Transactions on Games (2019).

[Ans+21] Ansari, Saba Gholizadeh, et al. "An Appraisal Transition System for Event-Driven Emotions in Agent-Based Player Experience Testing." *International Workshop on Engineering Multi-Agent Systems*. Springer, Cham, 2021.

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

[BY18] Bartocci, Ezio, Yliès Falcone, Adrian Francalanza, and Giles Reger. "Introduction to runtime verification." In Lectures on Runtime Verification, pp. 1-33. Springer, Cham, 2018.

[Ern+07] Ernst, Michael D., Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. "The Daikon system for dynamic detection of likely invariants." Science of computer programming 69, no. 1-3 (2007).

[Fer+21] Fernandes, Pedro M., Manuel Lopes, and Rui Prada. "Agents for automated user experience testing." *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2021.

[Her+17] Herzig, A., Lorini, E., Perrussel, L., Xiao, Z.: BDI logics for BDI architectures: old problems, new perspectives. KI-K¨unstliche Intelligenz 31(1) (2017).

[Hol+19] Christoffer Holmgård, Michael Cerny Green, Antonios Liapis, and Julian Togelius. Automated playtesting with procedural personas through MCTS with evolved heuristics. IEEE Transactions on Games 11, 4 (2018).

[kwe97] Stephen Kwek. 1997. On a simple depth-first search strategy for exploring unknown graphs. In Workshop on Algorithms and Data Structures. Springer.

[Mak+21] Makantasis, Konstantinos, Antonios Liapis, and Georgios N. Yannakakis. "The pixels and sounds of emotion: General-purpose representations of arousal in games." *IEEE Transactions on Affective Computing* (2021).

[Mel+21] Melhart, David, Antonios Liapis, and Georgios N. Yannakakis. "The Affect Game AnnotatIoN (AGAIN) dataset." *arXiv preprint arXiv:2104.02643* (2021).

[Min04] McMinn, P.: Search-based software test data generation: a survey. Software testing, Verification and reliability 14(2), 105–156 (2004).

[Mug+19] Luvneesh Mugrai, Fernando Silva, Christoffer Holmgård, and Julian Togelius.Automated playtesting of matching tile games. In 2019 IEEE Conference on Games (CoG). IEEE.

[Net+07] Dias Neto, A.C., Subramanyan, R., Vieira, M., Travassos, G.H.: A survey on model-based testing approaches: a systematic review. In: Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies (2007)

[Pra18] Prasetya, I. S. W. B. "Temporal algebraic query of test sequences." Journal of Systems and Software 136 (2018).

[Pra+20] ISWB Prasetya, Maurin Voshol, Tom Tanis, Adam Smits, Bram Smit, Jacco van Mourik, Menno Klunder, Frank Hoogmoed, Stijn Hinlopen, August van Casteren, et al. Navigation and exploration in 3D-game automated play testing. In Proceedings of the 11th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation. 2020.

[Pra+22] Wishnu Prasetya Fernando Pastor Ricos , Fitsum Kifetew, Davide Prandi, Samira Shirzadehhajimahmood , Tanja E. J. Vos, Karel Hovorka , Raihana Ferdous Fondazione Bruno Kessler, Angelo Susi Fondazione Bruno Kessler, Joseph Davidson. 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. 2022.

[SA06] Sen, K., Agha, G.: CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In: Int. Conference on Computer Aided Verification. Springer (2006)

[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.

[TOK19] J. Tuovenen, M. Oussalah, and P. Kostakos. 2019. MAuto: Automatic Mobile Game Testing Tool Using Image-Matching Based Approach. The Computer Games Journal 8, 3 (2019), 215–239. https://doi.org/10.1007/s40869-019-00087-z

[Wad92] Wadler, Philip. "The essence of functional programming." In Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 1992.

[Zar19] Imants Zarembo. 2019. Analysis of Artificial Intelligence Applications for Automated Testing of Video Games. In Proceedings of the 12th International Scientific and Practical Conference. Volume II, Vol. 170.

[Zhe+19] Yan Zheng, Xiaofei Xie, Ting Su, Lei Ma, Jianye Hao, Zhaopeng Meng, Yang Liu, Ruimin Shen, Yingfeng Chen, and Changjie Fan. Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning. In 34th International Conference on Automated Software Engineering (ASE). IEEE. 2019.