Intelligent Verification/Validation for XR Based Systems

Research and Innovation Action

Grant agreement no.: 856716

D2.2 – 2nd Prototype of the iv4XR Framework

Summary Document

iv4XR – WP2 – D2.2

Version 1.3

December 2021

Project Reference	EU H2020-ICT-2018-3 - 856716
Due Date	31/12/2020
Actual Date	29/12/2020
Document Author/s	Wishnu Prasetya (UU), Fernando Pastor (UPV), Rui Prada (INESC-ID)
Version	1.3
Dissemination level	Public
Status	Final

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	02/12/2021	Initial document structure and contents	WP
1.1	15/12/2021	Update Reporting section	FP
1.2	24/12/2021	Merging different parts	WP
1.3	29/12/2021	Final formatting and minor changes	RP

Document Quality Control			
Version QA	**Date**	**Comments (and if appropriate reason for change)**	**Initials of QA Person**
1.2	29/12/2021	Review and edits	IS

Document Authors and Quality Assurance Checks		
Author Initials	**Name of Author**	**Institution**
WP	Wishnu Prasetya	UU
IS	Ian Saunter	GWE
RP	Rui Prada	INESC-ID
FP	Fernando Pastor	UPV

TABLE OF CONTENTS

EXECUTIVE SUMMARY

The Deliverable D2.2 is the second prototype of the iv4XR Framework, which is an agent-based framework designed to enable automated testing of XR systems. This prototype is operational and has been tested with various internal case studies, though with respect to our final objective it is still incomplete. Since Deliverable-type D2.2 is of type OTHER, this document is not D2.2. itself, but rather a document that provides a tour on D2.2 and pointers to more detailed information on D2.2 technical details.

HOW TO READ THIS DOCUMENT

For the readers who prefer to try the Framework first, Section 1 below provides an overview where to find the Framework, build instructions, manuals, tutorials, etc.

The rest of the report is structured as follows.

- Section 2 presents the Framework's top-level architecture. The Framework's main components will be introduced here.
- Section 3 then briefly explains the key concepts of iv4xr agent-based testing. This section is essential as the iv4xr approach to testing is quite different. References will be given to where relevant papers providing more comprehensive explanation can be found.
- Section 4 briefly shows a concrete example of testing with iv4xr.
- Section 5 summarizes the WP2 status in terms of the completion status of its tasks.

1. WHERE TO GET D2.2 AND RELATED MATERIALS

D2.2 -- location	The current version of the Framework can be obtained from Github here: https://github.com/iv4xr-project/aplib Use e.g. the latest version of the Master-branch. The project linked above contains the Core part of the Framework. Tools from WP3 and WP4 are presented in the respective WPs. Their integration to the Framework is planned to take place in year-3.
D2.2 -- manuals	**Build Instruction for developers.** In the README.md: https://github.com/iv4xr-project/aplib#readme **Manuals explaining the concepts of iv4xr**: 1. The top of the README of the Framework (see the link above) contains an overview-list that links to different sections of documentation, including a link to a page explaining the concepts behind iv4xr agent programming. 2. **Paper:** *Aplib: Tactical Agents for Testing Computer Games*,

	by Prasetya, Dastani, Prada et al. Published in EMAS 2021, https://link.springer.com/chapter/10.1007/978-3-030-66534-0_2 3. The algorithm behind agents' execution loop is explained in this paper: *Aplib: Tactical Programming of Intelligent Agents*. Prasetya, 2019. https://arxiv.org/pdf/1911.04710 **Tutorials.** There are five tutorials on iv4xr agent programming and using test-agents. These are linked from its README: https://github.com/iv4xr-project/aplib#readme **API Reference.** This temporarily hosted here (waiting for a better solution, e.g. to be hosted in Jitpack): http://www.staff.science.uu.nl/~prase101/research/projects/iv4xr/aplib/apidocs/
D2.2 -- video	• We gave a demo of iv4xr in the Tool Demonstration track of ICST 2021 (awarded with the best presentation award). The video of the demo is here: https://zenodo.org/record/4661123#.YHVAsxMzZsM • TESTAR_iv4xr demo video that shows the sequence HTML report and the Analysis of an inferred state model. https://www.youtube.com/watch?v=hUyLWaHoOdI
D2.2 -- demo	A full example in the context of a realistic system can be found as part of the iv4xrDemo project that demonstrates the use of iv4xr to test a 3D maze-puzzle game. The Demo project can be found here: https://github.com/iv4xr-project/iv4xrDemo Look for example in the test-example src/test/java/agents/demo/RoomReachabilityTest.java.
D2.2 -- relevant papers	• *Aplib: Tactical Agents for Testing Computer Games*. Paper by Prasetya, Dastani, Prada et al. Published in 9th International Workshop on Engineering Multi-Agent Systems (EMAS) 2021. Springer. URL: https://link.springer.com/chapter/10.1007/978-3-030-66534-0_2 • *An Agent-based Architecture for AI-Enhanced Automated Testing for XR Systems*. Paper by Prasetya, Shirzadehhajimahmood, Ansari, et al. Published in the Workshop on Artificial Intelligence in Software Testing, co-located with ICST 2021. IEEE. URL: https://zenodo.org/record/4661945#.YbNcsr3MJ24

Table 1: *an overview of where to get D2.2, related readings and demo.*

2. ARCHITECTURE

The figure below shows the top-level architecture of the iv4xr Framework and its top-level workflow.

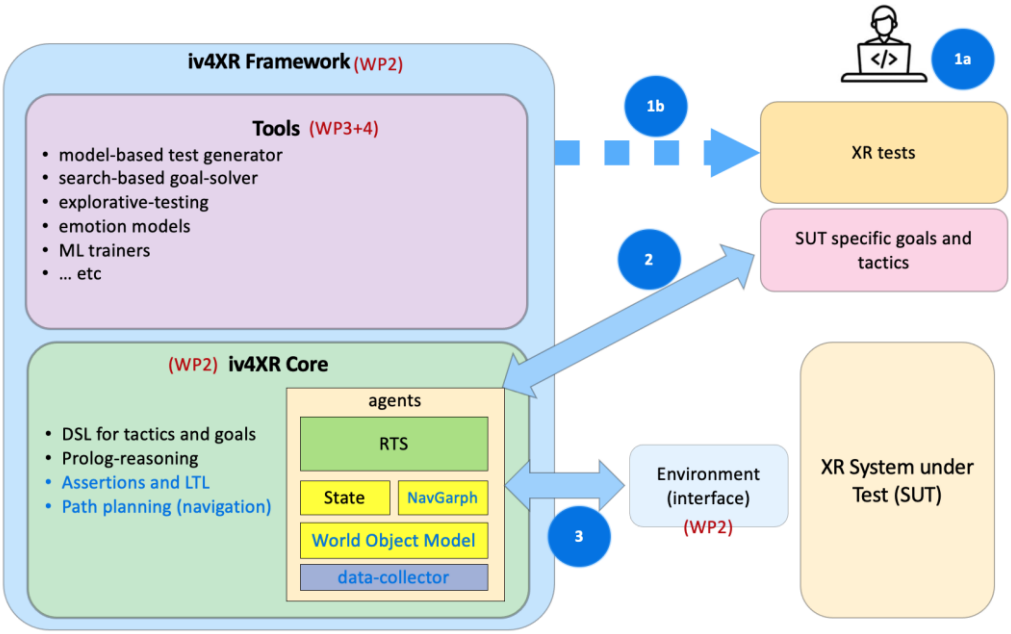

Figure 1: *the top-level architecture of the iv4XR Framework and its top level workflow*.

The Framework consists of the framework *Core* and various *tools* that use the Core. The Core provides the underlying agent-based concepts and programming for iv4xr. For example, it provides the implementation of agents and their runtime system, data collection and assertions for testing, basic agent reasoning through a Prolog engine and path planning, and also the embedded language for specifying tests (so-called Test Specification Language in Task 3.1 of WP3). Examples of tools are emotion models, explorative test agents[1], and model-based test generators. These tools are supplied by WP3 and WP4, while the Core is mainly built in WP2[2]. The design of the Framework as the integration of these components is led by WP2, while the actual integration of the tools into the Framework will be carried out in collaboration with the relevant WPs.

Iv4xr strives to remain neutral towards the XR technology used by the System under Test (SUT). This ensures as wide as possible usage of the Framework. In fact, our case studies in WP5 along with our various internal case studies all use different technologies, and we succeeded in targeting them all despite the diversity. The neutrality does mean that SUT developers must first implement

[1] This uses the tool TESTAR as an explorative agent. It targets the SUT through iv4xr's interfacing scheme and world representation. TESTAR is a background tool provided by the partner UPV for this project.
[2] Most Core is built in WP2, except the TSL part which is built in WP3.

an interface (called *Environment* in Figure 1) that would let iv4xr agents control the SUT. Additionally, a library of domain specific abstract/high-level actions need to be implemented as well (called domain specific goals and tactics in Figure 1 ---the term goal and tactic will be explained later). An example of such an action is the action to toggle an in-world switch. Depending on the SUT, this action may first require the agent to first approach the switch until it is close enough to it, and only then it can toggle the switch. Such a knowledge must come from the domain, as there is no way the agent could know this[3].

Figure 1 also shows the main workflow of iv4xr. The steps are indicated by circled numbers:

- Step ❶ (a/b): the XR developers write tests. Each test is expressed in terms of the aforementioned domain specific goals, specifying what should be tested (e.g. a switch *B* should be toggled, and this should cause some in-world lamp *D* to light up). In the basic form, a developer specifies the full sequence of goals that constitute a test (1a). In the more advanced form (1b), a tool from the Framework can be used to either generate a whole test suite, or to generate parts of a test (so the developer can leave out some subgoals, and a tool uses its algorithm to construct the missing subgoals). E.g. our model-based tool can generate a full test suite, if an EFSM model[4] of the SUT is provided. This is discussed in D3.3 of WP3.
- Step ❷: assigning the testing task to an agent. Every test is essentially a composition of aforementioned goals and tactics, structured together using the test specification language from Task 3.1 in WP3. Section 3 will show an example. To execute the test, this structure of goals and tactics is given to an agent.
- Step ❸: execution. When executing the goals and tactics given to it, the agent translates them to primitive interactions with the SUT. The agent does not directly interact with the SUT. Instead, it sends a command to the Environment component, which acts as an interface between the agent and the SUT (the reason for this indirection was mentioned earlier).

3. IV4XR AGENT-BASED SYSTEM: KEY CONCEPTS

In its core, the iv4xr Framework is an agent-based system. A testing task is formulated, essentially, as a pair of (*goal*,*tactic*) for a test-agent. The goal declaratively specifies **what** the task should accomplish, and the tactic is a program expressing a heuristic for accomplishing/solving the goal. There is a fundamental difference with the traditional way of testing, where a testing task is formulated as a step-by-step program formulating the sequence of steps that are involved for conducting the test. When testing a highly interactive system like an XR system, the sequence would consist of many interactions. A test that runs for 30 seconds can easily consist of over 1000 interactions. In contrast, tests on e.g. web or applications rarely exceed

[3] This side-steps the discussion that such a knowledge can perhaps be learned by the agent. The Framework actually integrates tools for learning. This is discussed in D3.3 of WP3.
[4] EFSM = Extended Finite State Machine. In adds variables to the usual FSM that the FSM's transitions can inspect and update. These variables can range over a finite or infinite domain.

50 interactions. XR systems also often behave non-deterministically or even adversariously (computer games and simulators are typical examples). So, a test program also needs to implement adaptivity heuristics to counter these effects. Programming this in the traditional way leads to a complex and cluttered codebase. If we insist on programming our test automation like this, in the long run this makes them expensive to make and maintain.

For example, consider a computer game whose interface offers these two primitive actions to interact with it:

```
attack()
moveToward(p)
```

The first action causes the player-character to attack an adjacent enemy. The second moves the player-character one frame-update towards a position *p*. Since only a **small** distance can be covered in a single frame update, multiple calls to this action are needed to actually reach the goal position *p*. A traditional test program to inspect some entity *e*, and deal with enemy threats along the way, would look like this:

```
while(current position != e.position) {
    if (there is enemy) attack()
    else moveToward(e.position)
```

afterwhich, it can inspect *e*. To add to the complication, if *e* is actually not immediately visible at the start, the test program has to be extended. The program would first need to steer the virtual character to explore the game world, until it can see e. This would require the addition of a loop. There may be more 'special cases' that need to be handled (e.g. the character might become stuck at some obstacle), and therefore the above loop has to be extended again. The test program above will then become very complex.

Iv4xr's agent-based approach provides a programming paradigm that is much better suited for programming control over highly interactive systems. An iv4xr agent executes in so-called *deliberation cycles*. At each cycle it decides which primitive action to execute:

```
repeat
    let A = set of available actions
    let A' = actions in A which are enabled in the current state
    decide which a ∈ A' to be chosen
    execute a
until temination-condition
```

Figure 2: *Each agent runs the above cycles of deliberation.*

Such a loop can drive a high rate of interactions. Changing situations can be rapidly handled by an action whose guard is tailored to identify the situations.

With such an execution model, in principle an agent can be programmed simply by giving it a set of actions, each can be guarded to decide its *enabledness* on a given state. The deliberation loop itself does **not** need to be programmed, since this is what the agent always does anyway[5], as seen in Figure 2. While the action-set does not need to be structured, because the actions' guards already determine what orders of executions are admissible, adding some degree of structuring improves their programmability. Iv4xr therefore offers a tactical language to allow primitive actions and the corresponding decision making to be composed in an abstract way. The same testing task shown before would look like this in iv4xr:

```
FIRSTof(
   attack().on(enemy is near),
   moveToward(e))
```

Extending this with exploration, when *e* is not immediately visible, amounts to just adding an exploration tactic to the **FIRSTof** construct. Similarly, if the agent can become stuck at some obstacle, an unstuck can also be added:

```
FIRSTof(
   unstuck().on(the agent is stuck),
   attack().on(enemy is near),
   moveToward(e).on(e is visible),
   explore() )
```

Figure 3: *an example of tactic composition with iv4xr DSL (Domain Specific Language).*

Goals

A tactic is sufficient for expressing a test that only involves navigating to and inspecting a single entity. It is not suitable for programming a test that requires interactions with multiple entities spread over a complex-shaped physical space, with the possibility of obstacles, hazards, or adversaries between them. This requires planning. To express planning in iv4xr we use a *goal-structure*, which is a composition of pairs (goal, tactic). Below is an example of such a plan for testing a world entity *e*. Imagine that in this world, access to *e* is not free, e.g., because it is located in an area that is guarded by a door/gate. So, to test *e* the agent would first need to open the door, which in turn requires interactions with entities b_1 and b_2: Here is then a test for *e*, expressed as a goal-structure:

```
SEQ(
   interactedWith(b1),
   interactedWith(b2),
   inspect(e) )
```

Figure 4: *an example of goal composition with iv4xr DSL (Domain Specific Language).*

[5] This action-based perspective of programming is similar to Back's Action Systems or Chandy-Misra's UNITY.

Here, interactedWith(b) is a pair of (G_b,T_b) where G is a goal specifying the desired state (b is interacted) and T is a tactic similar to Figure 3 to search for b in the virtual world, and interact with it.

Dynamic goals

While Figure 4 shows how a testing-task can be formulated as a high level plan, the tester does have to come up with the plan, e.g. to first interact with b_1 and b_2, before it can inspect e. It would be nice if the agent can come up with the plan on its own. This is possible and is discussed in WP3. The key feature from the Framework that enables this is the ability of the agent to dynamically deploy new subgoals. This allows e.g. an on-line heuristic to be programmed where the agent explores the virtual world and constructs plans as it goes. This is discussed in D3.3.

The architecture of an agent

Now that we have introduced the key concepts of agent-based testing with iv4xr, below we show the architecture of an agent:

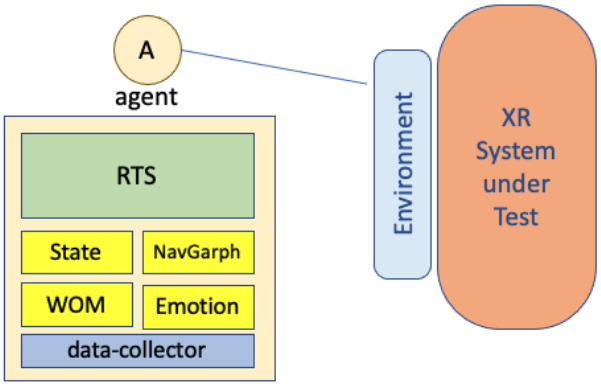

Figure 5: *the architecture of iv4xr agents. An agent has its own Runtime System (RTS). The yellow boxes are various state-like components that can be attached to it, e.g. a navigation graph and emption. A data collection can be attached to, to facilitate data collection for later analysis.*

To keep iv4xr neutral towards the technology used by the XR system under test (SUT), its agents do not directly interact with the SUT. Rather, they control the SUT through a Java class called *Environment*. This separates the Framework from whatever the technology used by the SUT, and hence the Framework does not depend on the latter. This does mean that in order to enable testing by iv4xr, SUT developers need to provide a concrete implementation of the class Environment.

An agent carries its own “run time system” (RTS), which is a module that allows the agent to accept and interpret goal structures. When the agent is run, the RTS essentially executes the deliberation cycles in Figure 2. It keeps track which goal from the goal structure that was given to it is the current goal to work on. When that goal is achieved it checks the goal structure to decide which goal is to go after next. If the current goal is not achieved yet, the RTS inspects its associated tactic to calculate which actions in the tactic are eligible for execution, and enabled in the current state (A’ in the algorithm in Figure 2). It then chooses one to be executed.

An agent also carries its own state. E.g. this state determines which actions from the tactic of the current goal are considered as executable. Different state components can be attached to the agent (shown yellow in Figure 5), and thus exposed to the goals and tactics. For example an agent may have a state component called “World Object Model” (**WOM**) representing the logical state of the SUT’s virtual world. It may also have a navigation graph as a state component, to help it plan routes to go to different places in the virtual world. It may have emotion (WP4) as a state component, and it may have just some “State” for its own bookkeeping. The last component of an agent is a data-collector to which the agent can report noteworthy observations (e.g. to report anomalies and violations found during a test run) or to write tracing information. Information collected can be immediately inspected or dumped to files for later data analyses.

Multi-agents

The Framework supports multi-agency. This means multiple agents can be deployed, and they can send messages to each other. This is useful for testing XR systems that allow multiple users. The feature is also useful for parallel testing. In this setup, we deploy multiple instances of the SUT and test agents, e.g. the latter can be used to explore different parts of the SUT’s state space, and hence speeding up the entire exploration.

Further reading

A more comprehensive explanation on iv4xr main concepts and agent programming can be found here:

1. *Aplib Concepts*. Documentation In the Framework. This can be found here: https://github.com/iv4xr-project/aplib/blob/master/docs/manual/aplibConcepts.md
2. *Aplib: Tactical Agents for Testing Computer Games*. Paper by Prasetya, Dastani, Prada et al. Published in EMAS 2021, https://link.springer.com/chapter/10.1007/978-3-030-66534-0_2
3. The algorithm behind agents' execution loop is explained in this paper: *Aplib: Tactical Programming of Intelligent Agents*. Prasetya, 2019. https://arxiv.org/pdf/1911.04710

4. HOW TO TEST WITH IV4XR

Figure 5 shows a simplified setup to test with iv4xr. We deploy an agent, giving it a goal structure that formulates a testing task as explained in Section 3. We attach this agent to an Environment that, in turn, is connected to the SUT. Note that the SUT developers must first implement a proper Environment, and additionally write a library of basic goals representing basic but high-level interactions on the SUT. To give a concrete example, the Figure 6 below shows a test on a 3D game called Lab Recruits. An IDE such as Eclipse can be used to write tests and run them. Figure 6 uses Eclipse. The SUT is visible in (1), and is being driven by the test. The testing task is formulated as a goal structure (marked by (2) in the Figure). In this case the structure is a SEQ-structure that takes a sequence of basic goals provided by a goal-library (called GoalLib in the Figure) as subgoals. For example, the first subgoal says that an in-world entity called *button1*

needs to be interacted first, whereas the third subgoal says that it expects that the property *isOpen* of an entity called *door1* should then be true.

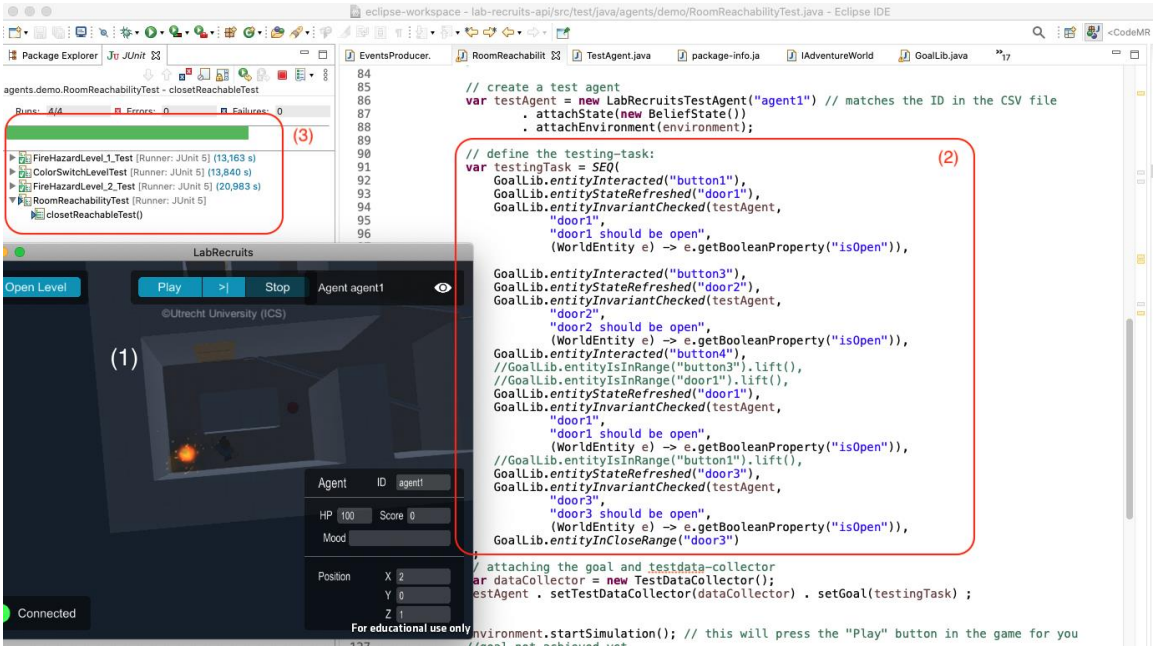

Figure 6: *testing with iv4XR using Eclipse IDE. The SUT is a 3D game, visible in (1). The testing task is formulated as a goal structure (2). In this case it is a sequence of basic goals provided by a goal-library (called GoalLib in the Figure). A summary of the tests (pass/fail) is shown in (3). Green means all tests succeed.*

More on how to test with a test agent can be found in our tutorials. These are linked from the README of the Framework. See: https://github.com/iv4xr-project/aplib.

5. D2.2 Tasks Status and Plan for D2.3

The table below gives an overview of the completion status of various tasks in WP2. Further explanation of these tasks is given in the subsections that follow.

☐ completed ☐ implemented, but need improvement ☐ planned

D2.2 status	Tasks and subtasks	D2.3 plan
Completed.	T2.1 Agent Runtime System	
Completed	T2.1 Multi-agent support	
To be improved.	T2.1 Integration with explorative functional test agents (WP3)	Iteration to improve its coherency and flexibility.
To do	T2.1 Integration with WP3 algorithms (e.g. search-based, reinforcement learning, model checking)	To be completed here.
To do	T2.1 Integration with emotion (WP4)	To be completed here.
To be improved.	T2.2 World Object Model	Iteration to improve its coherency and flexibility.
To be improved.	T2.3 Coupling Interface	Iteration to improve its coherency and flexibility.
Completed	T2.4 Information collection and reporting	
Completed	T2.4 Advanced information collection	

Table 2: *D2.2 status and plan for D2.3. explanations of these tasks are given in the subsections below.*

5.1 Task 2.1 Agent Runtime System

When created, each agent comes with its own instance of Runtime System (the RTS component in Figure 5). As mentioned, the RTS essentially implements the deliberation cycle in Figure 2. From a different perspective, the RTS interprets the goal structure *H* that is given to the agent. It keeps track which goal in H is its current goal. When that goal is achieved it checks H again to decide which goal is to go after next. If the current goal is not achieved yet, the RTS inspects its associated tactic to calculate which actions in the tactic that are eligible for execution and enabled in the current state (A' in the algorithm in Figure 2). It then chooses one to be executed.
Included in the Task 2.1 is providing the multi-agent capability mentioned in Section 2. We also add Integration to this Task, this is discussed separately in Section 5.4.

Completion status: see Table 2.

5.2 Task 2.2 and 2.3: World Object Model and Coupling Interface

The 'coupling interface' (Task 2.3) is the interface between agents and the SUT. This interface allows the agents to control the SUT and to observe its state. In iv4xr this interface is called Environment (the blue component in Figure 1 and Figure 5). Towards iv4xr, this interface needs to have a fixed structure, or else the agent does not know e.g. how to send a command to the SUT. On the other hand, it should allow the SUT's developers to implement this Environment in a way that is compatible with the technology of the SUT and the ontology of this SUT's virtual world (e.g. some worlds may have a concept of in-world pickable items, while other worlds do not

have this concept). We make use of Object Oriented inheritance to get the flexibility, shown in the class diagram below:

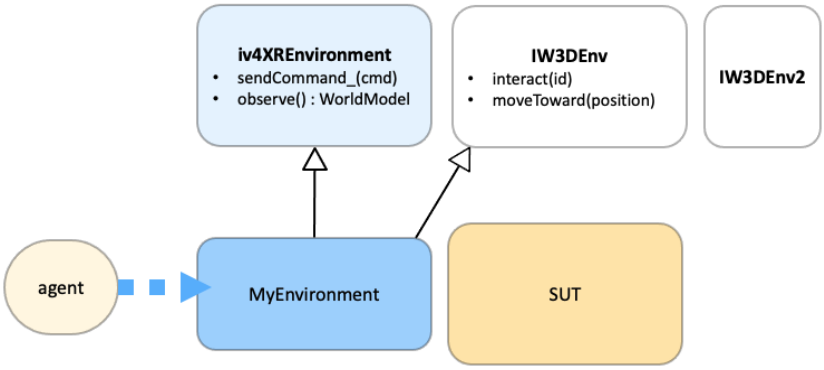

Figure 7: *the design pattern for implementing the Environment component.*

To implement their own Environment, SUT developers extend the pre-provided Iv4xrEnvironment class. This allows agents to assume that it gets an instance of Iv4xrEnvironment to interact with the SUT. E.g. the agents can then rely on the availability of the method observe() that would return the state of the SUT, encoded as a World Object Model.

The above interfacing scheme has been implemented. Additional interfaces such as IW3DEnv are provided, that offer the kind of methods common for interacting with certain types of XR systems. E.g. when the XR system has a 3D virtual world, then it is sensical to offer a method to allow the agent to travel in a straight line to some position p. The SUT developers can opt to implement such an interface over their instance of Environment.

Task 2.2 is about designing and implementing a common representation of the SUT's state. This representation is called *World Object Model* (also called WOM; and in the Framework it is called *WorldModel*). In Figure 1 and 5 it is indicated as one of the yellow (state-like) components.

Having a common representation allows agents to assume the same interface to e.g. obtain the state of some in-world entity in the SUT and to get the properties of this entity from this state. The representation is shown below (it is inspired by Domain Object Model, which is a common representation of web pages):

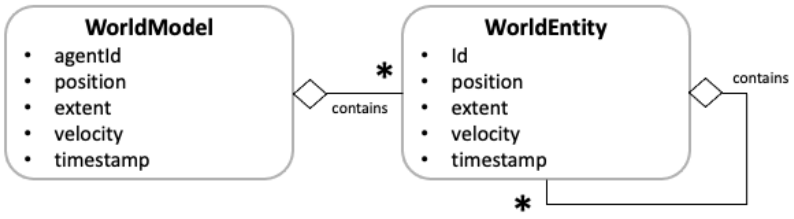

Figure 8: *a class diagram showing the structure of WorldModel (WOM). A WoldModel holds some typical properties of the agent that owns it, such as the agent's id and its in-world position. The WorldModel also contains a set of WorldEntities. They represent in-world entities currently known*

to the agent. Each such an entity is represented by its basic properties, such as its id and position, and may furthermore contain sub-entities (e.g. if it is an in-world control panel, which may in turn consist of several switches).

Completion status: see Table 2.

5.3 TASK 2.4: REPORTING

This task is meant to build a module that allows a test agent to report its finding, e.g. violations of assertions and events that signify the covering of states of interest (useful for giving an indication of the completeness level of the test, e.g. whether or not all pre-defined states of interest are covered). The module is implemented. It is the *data-collector* component in Figure 1 and Figure 5. This data-collector can actually do more than just reporting violations. It can do so-called *instrumentation*. It allows a function to be attached, which maps the SUT state (e.g. in terms of WorldModel) to a set of numeric values. The function is sampled at every cycle in Figure 2. The sampled data iscollected and can be saved to files which later can be subjected to data visualization and analyses. Several standard visualization scripts are provided, e.g. to produce time-graphs and heatmaps.

TESTAR is an iv4xr framework tool that acts as an exploratory Functional Test Agent (FTA). TESTAR launches and connects with the SUT using an additional UIAutomation API, something that allows the tool to detect the Graphical User Interface (GUI) in Windows environments, take screenshots of the SUT and create visual reports.

As TESTAR explores the SUT (D3.3 contains more information about TESTAR exploratory cycle), it creates an action-by-action HTML report and infers a state model. This tool has also been extended with a *listening* implementation in order to allow the integration of goal-solving agents and the inference of state models. Figure 9 is an extension of Figure 6 that shows the integration of goal structures in TESTAR. Figure 10 shows the state model visualization.

Section Where to get D2.2 and Related Materials contains a TESTAR video that shows the HTML report and analysis of the state model.

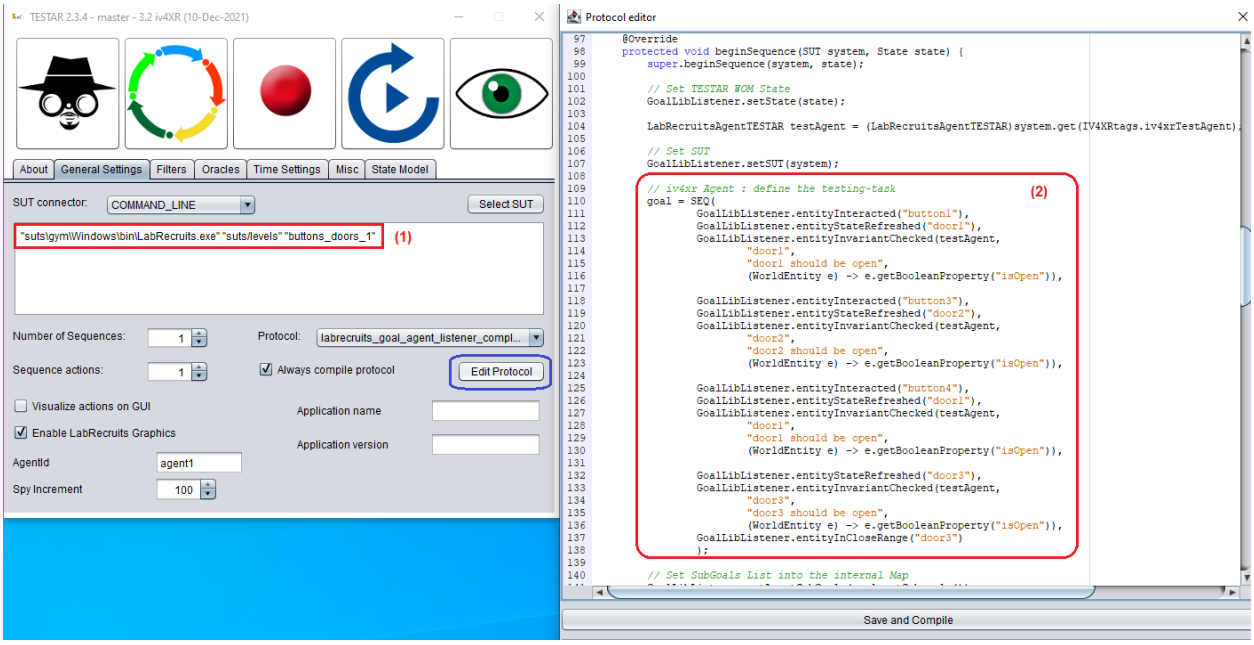

Figure 9: *TESTAR tool integrated with goal solving agents. The SUT path and level to test needs to be selected in the SUT connector field (1). The desired sequence of goals should be created in the beginSequence method of TESTAR (2).*

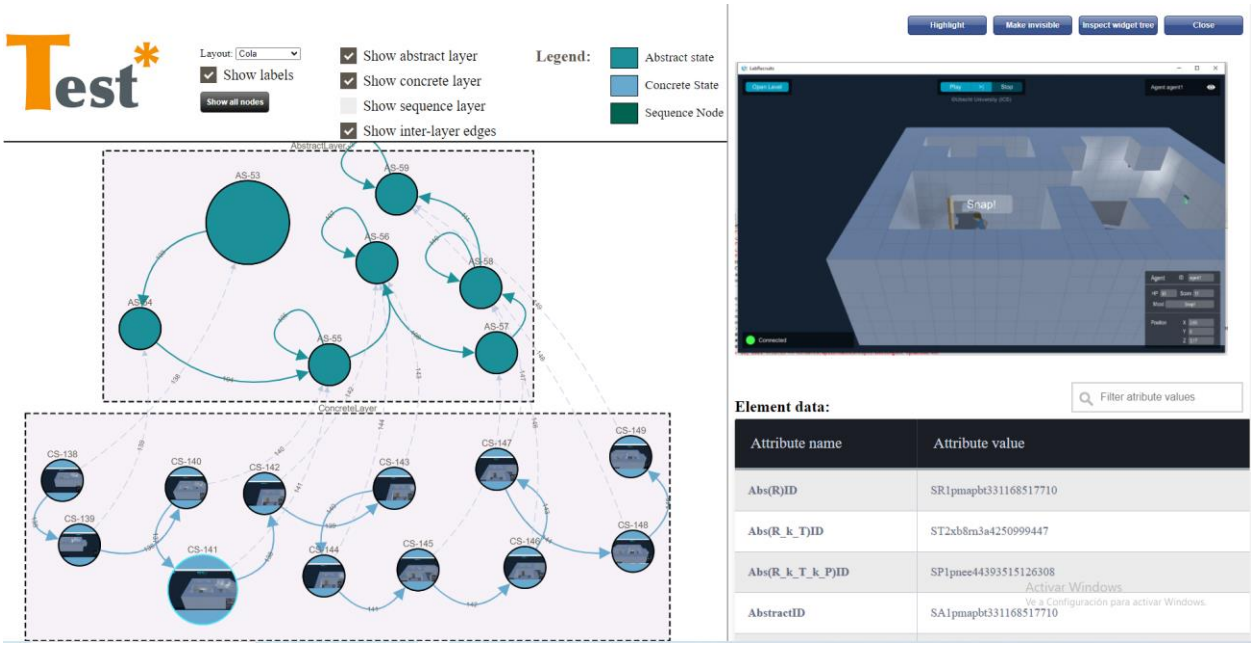

Figure 10: *This image shows the usage of the Analysis mode by the TESTAR tool in order to visualize the information inferred in the state model. These models contain information about the states, entities and actions discovered in the iv4xr SUT.*

Although this task has been completed, throughout the last year of the project, maintenance and possible improvements to the reporting system are expected.

Completion status: see Table 2.

5.4 INTEGRATION

Ultimately, various tools produced by WP3 and WP4 need to be included into the Framework (see the architecture in Figure 1). This means that they need to be integrated into the Framework, which also implies that a sensical integration scheme should be prepared. We add this integration task to WP2 under Task 2.1, in collaboration with WP3 and WP4. Currently the project has been working on the major components separately (e.g. the Core, and each of the tools in Figure 1). The actual integration work will be carried out together with WP3 and WP4. This integration is scheduled in Year-3 of the project.

Completion status: see Table 2.