Intelligent Verification/Validation for XR Based Systems

Research and Innovation Action

Grant agreement no.: 856716

D3.1 – Test Specification Language

iv4XR – WP3 – D3.1

Version 2.1

December 2020

Project Reference	EU H2020-ICT-2018-3 - 856716
Due Date	31/12/2020
Actual Date	30/12/2021
Document Author/s	Tanja Vos (UPV), Wishnu Prasetya (UU), Rui Prada (INESC-ID), Angelo Susi (FBK)
Version	2.1
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	6/11/2020	Initial document structure and contents	WP, TV
1.1	10/12/2020	Draft	WP
1.2	24/12/2020	Revised draft after QA comments	WP
1.3	30/12/2020	Final arrangements for submission	RP
2.0	19/12/2021	Initial setup for version 2	WP
2.1	28/12/2021	Updated content	WP

Document Quality Control			
Version QA	**Date**	**Comments (and if appropriate reason for change)**	**Initials of QA Person**
1.1	17/12/2020	Comments and minor edits	AS
1.1	21/21/2020	Comments and minor edits	RP
2.1	29/12/2021	Comments and minor edits	IS
2.1	29/12/2021	Minor edits	AS

Document Authors and Quality Assurance Checks		
Author Initials	**Name of Author**	**Institution**
TV	Tanja Vos	UPV
RP	Rui Prada	INESC-ID
WP	Wishnu Prasetya	UU
AS	Angelo Susi	FBK
IS	Ian Saunter	GWE

Executive Summary

This deliverable D3.1 presents iv4XR Test Specification Language (TSL), which is provided as an embedded DSL in Java. The TSL will allow developers to abstractly formulate testing tasks, to be automatically carried out by test agents. The purpose of this document is to give a summary of the concept behind this TSL. Note that the deliverable itself (D3.1) is classified of type OTHER, which concretely is provided as modules within iv4XR Framework. The purpose of this document is to provide a summary of what the deliverable is about, without having to go through the source code of the modules.

Acronyms and Abbreviations

DSL	Domain Specific Language
TSL	Test Specification Language

1. Introduction

Figure 1 below shows the main workflow of testing with iv4xr. To test an XR system, a tester or developer formulates the tests. In iv4xr's terminology these are called "testing tasks". These tasks are then given to one or more test-agents that will then carry them out. An agent converts each testing task given to it into a series of interactions with the XR system under test (SUT) to drive it to certain states and to check some or all of these states for their correctness.

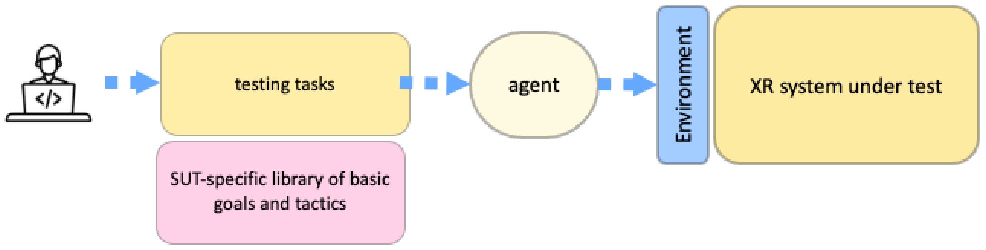

Figure 1: *Main workflow of testing with iv4XR.*

An important part of the iv4xr Framework is a so-called *Test Specification Language* (TSL) to allow testers/developers to declaratively express testing tasks. This language has been implemented as an embedded Domain Specific Language and is provided as part of the iv4xr Framework. We will later provide pointers to where the implementation can be found. In this document we will give a summary of the underlying concepts of this language, and some examples to illustrate the use of this language.

The rest of this document is structured as follows:

Section 2	This introduces the basic concepts of testing tasks through an example of a simple task.
Section 3	This section discusses how to keep testing tasks abstract.
Section 4	This section discusses TSL constructs for formulating complex tasks.
Section 5	This section discusses some key implementation aspects of TSL.
Section 6	This section gives pointers where to find D3.1 results.
Section 7	This section describes our plan with TSL in the remaining duration of the project..

2. Specifying Simple Testing Tasks

In the simplest form a testing task can be seen as an implicative formula of the form:

(1) $\varphi \rightarrow \psi$

This specifies the task to check that all states[1] of an XR system that satisfy φ should also satisfy ψ. The ψ-part is also called "*invariant*" or "assertion". It represents a correctness constraint that all states in φ are expected to satisfy. To show how this is done let us consider the following example.

Figure 2 below shows a small level in a 3D puzzle game called Lab Recruits[2]. Access to different rooms in the game is guarded by doors, which in turn can be opened/closed by toggling the correct in-game buttons. The player's starting position is indicated by the blue circle.

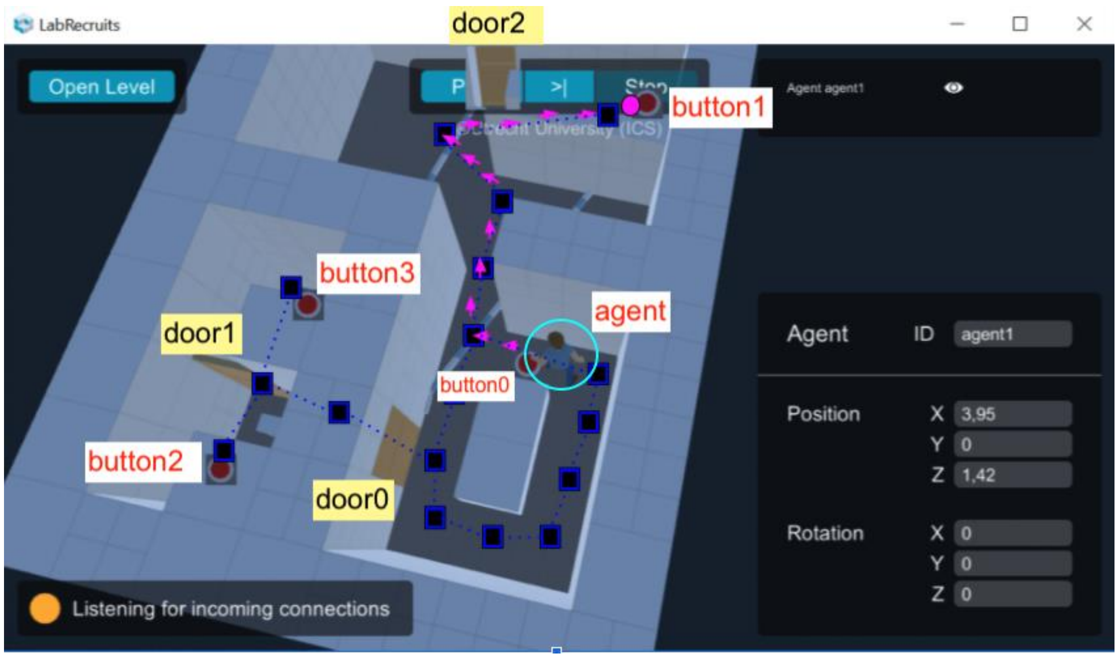

Figure 2*: a small level in Lab Recruits, with 4 buttons and three doors. The starting position of the agent is shown in the blue circle.*

Consider a simple testing task is to verify that the initial state of button1 is set to "off". Checking the state of an in-game entity requires however that the entity is *visible* to the test-agent. So the testing task can be expressed in the format of (1) informally as follows:

(2) "button1 is visible" $\rightarrow$ "button1 in off"

[1] Since our verification approach is through **testing**, literally checking "all" states would not be possible, since there will be infinitely many of them. In testing we would then have to sample the states. Note that just randomly sampling the states is not going to work. Instead, we need to sample relevant states (states that satisfy φ).

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

Note that at the start the agent **cannot** actually see the button due to a wall that blocks the agent's sight. So, the above 'implication' is immediately true. But indeed this does not actually check the intended correctness property. In general, an implication is trivially true on the states where the antecedent is false. Checking the invariant part of such states is pointless. To make the test meaningful we should actually formulate the implication as a sequence: first the agent should explore the level until it gets to a state where the antecedent is true (in the above case: until button1 is visible), and then it can check the invariant-part of the task. So concretely, the task is formulated as follows in iv4xr using the TSL, where a sequencing operator "**SEQ**" is now used instead of implication:

```
var testingTask = SEQ(
      entityStateRefreshed("button1"),
      entityInvariantChecked("button1",
            (WorldEntity e) → ! e.getBooleanProperty("isOn")),
  )
```

Figure 2: *a simple testing task to check the initial state of button1 in the Lab Recruits level shown in Figure 1.*

The blue part above captures the φ-part of the task, and the yellow part formulates the invariant to check.

3. ABSTRACT TEST

Testers should be able to formulate testing tasks abstractly, because then less time and effort are needed to construct them. In the example shown in Figure 2, the φ-part requires the agent to find a state where button1 is visible (it is phrased as "the entity state is refreshed" by the task). While the formulation of the subtask is simple, carrying it out is **not** simple. The button is not initially visible to the agent, so it first needs to explore the level. This requires many primitive interactions to command the agent to travel to different positions, not to mention the calculation of which positions should the agent explore, and in which order should they be explored. Obviously, we want such details to be automated (to manually programme them every time would blow up the cost of testing).

To allow the details on how tasks are to be solved to be hidden from the testers, the iv4xr agent programming approach represents a task, in its simplest form, as a pair (g,t) where g is a so-called **goal** and t is the corresponding **tactic**. A goal expresses the states that are desired, and a tactic is a heuristic for the agent to drive itself to a state satisfying g. If we already have a library of basic tactics (the red component in Figure 1), a tester only needs to specify what the goal of a task or subtask is, and which basic tactics are to be used to achieve/solve the goal (and hence to complete the task).

For example, suppose we have these as basic tactics: (1) *navigate*(e) calculates a path to an entity e, and then drives the agent to follow this path, and (2) *explore*() drives the agent to explore to unseen (but reachable) parts of the SUT's virtual world. The example subtask *entityStateRefreshed*("button1") in Figure 2, can now be formulated as follows:

```
goal("some name")
  . toSolve(state → state.time == state.getEntity("button1").time)³
  . withTactic(explore())
```

Figure 3: *the implementation of the task entityStateRefreshed("button1").*

Using *explore*() to solve the goal works because initially the button is not known to the agent. Unfortunately, if for some reason the agent wants to check the button's state again, then just using *explore*() will not work (because the button is no longer unexplored). The following shows how basic tactics can be combined to form a more powerful heuristic:

```
FIRSTof(navigateTo("button1"),
          explore()                                                              ,
          ABORT())
```

Figure 4: *An improved tactic for the task in Figure 3.*

The **FIRSTof** combinator constructs a new tactic that chooses the first enabled action of its sub-tactics. The tactic *navigateTo*() is only enabled if a path to button1 can be calculated (e.g. the purple path in Figure 1). If the button is already known (the agent has seen it), this path can be calculated, and *navigateTo*() will be the tactic used to guide the agent to the button. However, such a path does not exist at the beginning (because the agent has not seen the button yet), in which case the tactic above then falls back to explore(). If neither *navigateTo*() nor *explore*() is possible, the button must thus be unreachable (e.g. it could be behind a closed door), and the tactic above falls back to ABORT(), which would abort the corresponding goal[4].

Assuming a set of basic tactics are given, the above approach means that the tester only needs to specify which composition of basic tactics he/she wants to use. Furthermore, we do not have to repeat the formulation for goals that are similar. For example, the goal and tactic shown in Figure 3 and 4 can be generalized for every entity *e* and formulated as a method *entityStateRefreshed*(e) which can be reused for refreshing the agent's belief on the state of *any* entity *e*.

[3] In this example, the time of an entity refers to the last time it was observed, whereas the time of a state refers to the time of the most recent observation on the SUT. So the formula says that button1 was observed in the most recent observation.

[4] A higher level task then has to decide whether to give up as well, or to try a different goal, e.g. opening a door.

The mentioned collection of basic tactics is however quite domain specific[5], so the Framework cannot provide it. The SUT developers should provide it. However, it is a one-off investment. The tactics can then be reused for automating any number of testing tasks targeting the SUT. In WP3 we do work on reusable components that can be used across different SUTs, e.g. modules with path finding algorithms, or modules of model-based goal solvers.

4. COMPLEX TESTING TASKS

Note first that since in iv4xr a task is represented as a goal (Section 3 above), for an iv4xr agent "executing a task" equates to "**solving** a goal", or, we also say "solving a task".

Now, consider again the example game-level in Figure 2. Suppose that the level objective is to escape through door2. To verify that the level is playable, we need to check whether the state where the door is open is reachable. In the $\varphi \rightarrow \psi$ form as in (1) this can be expressed as this testing task:

(3) "door2 is open" $\rightarrow \psi$

where ψ is some invariant to check, e.g. it could assert that the point collected by the agent should be at least some value *p*. It could be simply "true" if we actually just want to check that the state "door2 is open" is reachable.

However, unlike the previous example in Section 2, this task is more complicated to solve. Door2 is initially closed. To open it, button2 has to be toggled. However, the button is behind a closed door (door0), so the latter must first be opened. Furthermore, if the agent manages to reach button2, toggling it will close door0 again, causing the agent to become trapped, so the agent also needs to toggle yet another button to leave the trap. So, solving the task (3) requires the agent to toggle buttons and passing doors in a specific order. Finding the solution is not trivial for the agent. For example, at the start the agent does not even know that button2 and button3 exist (because they are not visible to the agent). Moreover, the state space would appear to the agent as a much denser space where it is far from obvious what it should do to bring it closer to the goal state.

In WP3 we investigate algorithms (solvers) that can solve tasks automatically, under certain assumptions[6], e.g. if a behavioral model of the SUT is given. For the cases when the task at hand is too hard for the agent (and its solvers) to solve, TSL offers a way to express a task as a composition of subgoals. These subgoals can be seen providing waypoints towards solving the final goal. An example is illustrated in the picture in Figure 5, where a testing task $\varphi \rightarrow \psi$ is refined to have three subgoals T_1, T_2, T_3 to help the agent to solve the φ part.

[5] For example, in the example game in Figure 2, a button needs to be toggled to open an in-world door. In a different world, opening a door may require the presence of a key in the agent's inventory. Obviously, this is a quite different mechanic of opening doors. Even moving around can be very different. Driving an agent that can fly in all 3D-direction, is very different than driving an agent that can only walk on a surface.

[6] No general goal solver exists though, due to the Halting problem.

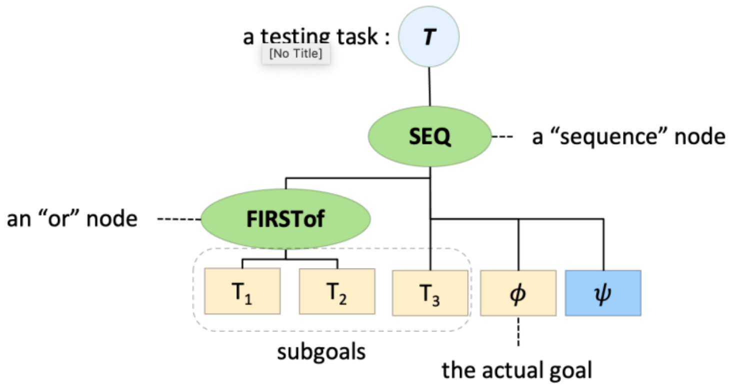

Figure 5: *A visualisation of a testing task $\varphi \rightarrow \psi$. Imagine that φ is hard for the agent to solve directly. In the above task we add three subgoals T_1, T_2, and T_3. The goal structure above directs the agent to first solve either T_1 or T_2, and then continue with T_3 , and then φ and ψ.*

In our TSL, the testing task in Figure 5 is formulated as a so-called goal-structure like this:

```
task =  SEQ(FIRSTof(T1 ,T2),
                T3 ,
                φ
                invariantChecked(ψ) )
```

A goal-structure is either a goal, or a structure that is recursively composed from smaller goal-structures. If T_1,...,T_n are goal-structures, the construct **SEQ**(T_1,...,T_n) is a goal-structure that is solved when all the sub-goals T_1,...,T_n are solved, and moreover they are solved in the order as given in the sequence. The construct **FIRSTof**(T_1,...,T_n) is a goal-structure that is solved when one of the sub-goals T_1,...,T_n is solved. These subgoals will be tried one at a time in the order as given by the sequence, up until one that succeeds is found. More on the TSL, including more language constructs such as REPEAT, is described in the paper listed in Section 6; the paper has been presented in the International Workshop on Engineering Multi-Agent Systems (EMAS).

As a concrete example, the testing task (3) at the beginning of this section (about verifying that the level in Figure 2 is playable) can be refined to the task shown in Figure 6 below, formulated in our TSL.

```
var testingTask = SEQ(
        entityInteracted("button1"),
        entityStateRefreshed("door0"),
        entityInvariantChecked("door0",
              (WorldEntity e) -> e.getBooleanProperty("isOpen")),
       entityInteracted("button2"),
       entityStateRefreshed("door1"),
       entityInvariantChecked("door1",
            (WorldEntity e) -> e.getBooleanProperty("isOpen")),
    entityInteracted("button3"),
    entityStateRefreshed("door0"),
    entityInvariantChecked(..."door0",
            (WorldEntity e) -> e.getBooleanProperty("isOpen")),
   entityStateRefreshed("door2"),
   entityInvariantChecked("door2",
            (WorldEntity e) -> e.getBooleanProperty("isOpen"))
 );
```

Figure 6: *A testing task to verify that the game-level in Figure 1 is playable. This comes down to proving that the state where door2 is open is reachable. Opening the door involves solving a puzzle, where the buttons and doors in the level has to be toggled and passed in a specific order. The goal-structure above captures the solution. The yellow parts are invariants added expressing that we expect the corresponding door to be open after we toggle the button that should open them. The last invariant is the most important one, as it is that check that door2 is open.*

A preliminary study on the usability of TSL is still on-going, e.g. to assess whether the current language constructs are expressive enough to express the kind of testing tasks needed by our Industry partners. Another important research question that this study will address is the robustness/adaptivity of iv4xr automation. For example, suppose a developer has invested effort to write a goal structure *U* to automate a testing task *T*. During its development, we can expect an XR system to be changed frequently. We do not keep fixing *U* every time the system under test is changed, especially if the change is either superficial or has no direct relation with the task *T* itself. In other words, we want *U* to be robust. Assessing this is still an on-going study.

5. Implementation

Rather than offering TSL as its own native language, we will instead provide it as an embedded Domain Specific Language (DSL). A DSL is a programming language or an expression language used to formulate constructs in a certain domain, as opposed to a general purpose language, like Java, that is intended to be used to solve problems in any sort of domain (at least by intent). HTML is an example of a simple DSL, and SQL is a more elaborate example. Having TSL as a native DSL (the way e.g. SQL is) would offer better programming experience for developers, but it is also prohibitively expensive to develop and maintain. Moreover, without enough support tools (e.g. static type checker, debugger, documentation tool) not much companies will want to use it. We therefore choose to offer TSL as an “embedded” DSL instead. An embedded DSL embeds the DSL in some host language. Languages like Java, Haskell, or Python are commonly used as hosts for embedded DSLs. Rather than offering its own native language constructs, an embedded DSL offers a set of APIs that are crafted in such a way to mimic actual language constructs. We have embedded TSL in Java, which is a popular language in XR development. An embedded DSL retains much of the fluency of a native DSL, but the trade off is that programmers would still be limited by the syntax of the host language. On the other hand, the DSL’s users also get access to the host language's (in our case Java) full features and tooling. Examples of DSLs that exploit this approach are: the tactic DSL in the domain of theorem proving, a DSL for generating mocks in software testing, and DSLs for building parsers.

6. D3.1 RESULTS

Below we give an overview of our D3.1 outputs:

1. **Implementation**. TSL is delivered as a part of the iv4xr Agent Framework. The Framework is available in the link below:

   https://github.com/iv4xr-project/aplib

   Key classes for TSL are AplibEDSL and Iv4xrEDSL in the package nl.uu.cs.aplib and eu.iv4xr.framework.

2. **API Reference.** It can be found here, see e.g. the key classes AplibEDSL and Iv4xrEDSL mentioned above.

   https://webspace.science.uu.nl/~prase101/research/projects/iv4xr/aplib/apidocs/

3. **Paper** presenting the main concepts of this TSL has been presented in the 8th International Workshop on Engineering Multi-Agent Systems (EMAS) 2020: *Tactical Agents for Testing Computer Games*, Prasetya, Dastani, Prada, et al.: https://emas2020.in.tu-clausthal.de/files/emas/papers-h/EMAS2020_paper_6.pdf

4. **Video**: https://screencast-o-matic.com/watch/cYfT0QApOT

7. PLAN

We plan to finish our preliminary assessment (as mentioned Section 4) of TSL. This will be done using several internal demonstration projects[7]. For the remaining duration of the project, we will also support WP5 in using TSL in WP5's pilots.

[7] See for example: https://github.com/iv4xr-project/iv4xrDemo