

An Agent-based Architecture for AI-Enhanced Automated Testing for XR Systems, a Short Paper

I. S. W. B. Prasetya
Utrecht University, the Netherlands
s.w.b.prasetya@uu.nl
Orcid: 0000-0002-3421-4635

Samira Shirzadehhajimahmood
Utrecht University, the Netherlands
s.shirzadehhajimahmood@uu.nl
Orcid: 0000-0002-5148-3685

Saba Gholizadeh Ansari
Utrecht University, the Netherlands
s.gholizadehansari@uu.nl
Orcid: 0000-0002-7135-5605

Pedro Fernandes
INESC-ID and Instituto Superior Técnico, Univ. de Lisboa
Portugal, pedro.fernandes@gaips.inesc-id.pt
Orcid: 0000-0002-2840-562X

Rui Prada
INESC-ID and Instituto Superior Técnico, Univ. de Lisboa
Portugal, rui.prada@tecnico.ulisboa.pt
Orcid: 0000-0002-5370-1893

NOTE: This is a preprint of an article with the same title, published the proceedings of the 1st International Workshop on Artificial Intelligence in Software Testing, which is bundled in 2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW) proceedings. The paper is published by IEEE, DOI: 10.1109/ICSTW52544.2021.00044

Abstract—This short paper presents an architectural overview of an agent-based framework called iv4XR for automated testing that is currently under development by an H2020 project with the same name. The framework’s intended main use case of is testing the family of Extended Reality (XR) based systems (e.g. 3D games, VR sytems, AR systems), though the approach can indeed be adapted to target other types of interactive systems. The framework is unique in that it is an agent-based system. Agents are inherently reactive, and therefore are arguably a natural match to deal with interactive systems. Moreover, it is also a natural vessel for mounting and combining different AI capabilities, e.g. reasoning, navigation, and learning.

Index Terms—AI for automated testing, automated testing XR systems, agent based testing, AI for testing games

I. INTRODUCTION

The iv4XR Framework¹ is an open source agent-based framework for automated testing of ‘Extended Reality’ based systems. This subfamily of interactive systems includes 3D games, 3D simulations, VR systems, and AR (Augmented Reality) systems. The domain urgently needs test automation support as manual testing is becoming very expensive and tool support is scarce (even record and replay is often not available). An *agent* is essentially a program that interacts with an environment by repetitively performing actions, either on its own initiative or as reaction to events generated by the environment. An agent is thus inherently a *reactive program*, and in this sense it is fundamentally different than e.g. a

procedure or a service. Arguably, this makes agents a more natural framework for testing interactive systems.

The iv4XR Framework is currently under development by an H2020 project with the same name. It has reached a working prototype level, and is undergoing internal piloting. To deal with the huge and fine grained interaction space of XR systems, iv4XR necessarily relies on AI. It has however its own, rather unique, perspective of how AI is to be deployed to aid software testing.

While the current interest in AI mainly focuses in machine learning, iv4XR’s main AI is agent-based AI. The agent community has long insisted that intelligent agent is AI, a position that is also supported by the AI community [1], [2], [3]. Iv4XR is inspired by a popular concept of intelligent agents called *Belief-Desire-Intent* (BDI) [4], [5], [6]. An agent is thought to have a mental state, which includes its ‘beliefs’ and ‘desires’. ‘Desires’ are formulated as goals that the agent seeks to achieve. ‘Intent’ represents the agent’s plan towards achieving a goal; this corresponds to the concept of function or method in traditional programming. The intelligent part comes from the agent’s ability to reason, e.g. through reasoning rules, about its believes and goals, to decide which goal to pursue, and which plan to use. This approach combines a *proactive dimension*, that together with the *reactive nature* of agents provides more versatility and strength for automated testing than just invoking e.g. a random tester or a genetic algorithm.

‘Belief’ is also different than the traditional concept of ‘state’. A BDI system acknowledges that belief is not necessarily the same as the reality. For example, suppose a test agent wants to test if a certain button in the UI of some

This work is supported by EU Horizon 2020 research and innovation programme, grant 856716 project iv4XR (Intelligent Verification/Validation for Extended Reality Based Systems) and by national funds through FCT, Fundação para a Ciência e a Tecnologia, project UIDB/50021/2020

¹<https://github.com/iv4xr-project/aplib>

system under test (SUT) behaves correctly. To find the button, it might *believe* that the button is located in e.g. *panel_x*. By acknowledging that this is only a belief, and not necessarily a fact, we would be more compelled to program what the agent should do if the belief turns out not to hold, e.g. if the developer has moved the button to *panel_y*. This mindset encourages the development of robust test strategies.

Agent-based AI and machine learning are not mutually exclusive. For example, reinforcement learning can be seen as an approach for finding a policy for an agent towards solving a goal. This can be leveraged by combining it with BDI agent’s ability to reason. Reasoning rules improve the agent’s ability to discern which actions in a given state are much more likely, or else unlikely, to lead to worthy rewards towards reaching the goal, hence pruning the space that the agent has to explore to learn. Conversely, when not all reasoning rules are known, techniques such as rules learning can be used to learn them.

Iv4XR has more features than just BDI, e.g. goal-based and tactical programming [7], automated navigation and exploration, and integration with other testing tools such as TESTAR [8]. A demonstration is available, where iv4XR is used to automate the testing of a 3D game called Lab Recruits². Larger pilots are work in progress.

This short paper presents an architectural overview of the iv4XR Framework. Section II introduces the key concepts, the architecture of a test agent, and explains how it works. Section III explains iv4XR’s design pattern to make it extensible for targeting an arbitrary SUT. Section IV discusses some related work. Section V concludes and mentions future work.

II. TESTING WITH IV4XR

Iv4XR is implemented as a Java library. This also means that Java is the language to use when formulating testing tasks—the advantage is that Java is a popular language, well supported with development tools. Iv4XR also comes with a set of APIs that mimic a Domain Specific Language (DSL), allowing testers to formulate tasks more fluently. Since it is not possible to have one automated testing framework to cater all sorts of technologies and ontologies of the target systems, iv4XR is designed from the outset to be extensible, relying on two Java features to do this: *inheritance* to allow testers to customize standard behavior, and *λ-expression* to allow new behavior to be fluently passed as parameters.

Figure 1 shows a top level architecture of test agents. But let us first explain iv4XR’s concept of ‘automated testing’.

A. What should we automate?

Given a System under Test (SUT), iv4XR facilitates the development of strategies which can be used to automatically ‘solve’ testing tasks. Strategies might be SUT-specific, but once developed they can be reused to automate all sorts of testing tasks for the SUT (or its family). While testing tasks can be automatically solved, iv4XR does expect the tester to specify what the tasks are. This is *different* than automation

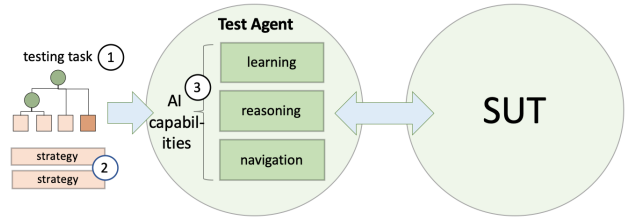


Fig. 1. An iv4XR test agent accepts testing tasks. Strategies are used to solve them, which in turn make use of general AI capabilities such as reasoning and navigation.

provided by tools like QuickCheck, T3, or Evosuite [9], [10], [11] which only need the SUT to be given, after which they can generate tests. Such an approach works when testing units (e.g. a method or a class). An interactive system is however more complicated. The state space is much larger, and has a complicated structure. Without being more specific in *what* we want to test, just wandering around trying different things is unlikely to be effective, which is why iv4XR is aimed towards task-level automation.

The simplest form of ‘testing tasks’ is the task to verify an ‘assertion’. Two common types of assertion-like properties are these (borrowing CTL notation [12]):

- *Existential* $EF\phi$, asserting the existence of an execution that leads to a state satisfying ϕ . As an example: in a web application a page named *Purchase* should be reachable.
- *Universal* $AG(\phi \rightarrow \psi)$, asserting that all states satisfying ϕ should also satisfy ψ . For example, we might require that the aforementioned page *Purchase* (the ϕ part) contains a button named *cancel*, and clicking on it should empty the user’s purchasing basket (the ψ part).

In both cases, a testing algorithm will have to find at least one right sequence of interactions that would move the SUT to ϕ . This part is often very challenging, whereas checking the ψ part, in the case of a universal assertion, is usually straight forward. For example, if the SUT is a computer game, and ϕ is a key room in the game, verifying $EF\phi$ would effectively require an algorithm that knows how to play the game, at least as far as getting itself to reaching the room. Obviously, this is not an easy feat. From this perspective, ‘strategies’ pointed in Fig. 1 are, in the heart, typically aimed to guide the agent towards different kinds of ϕ .

B. Test agent

Figure 1 shows the architecture of a test agent. The agent controls the SUT by executing a series of *actions*. Actions can be expected to be very *primitive*. E.g. if the SUT is a computer game, an ‘action’ could be to move an in-game character in a given direction for few frame updates. Modern games can run at the rate of 100 frame updates/second, so such an action will only move the character for a very small distance. Note that having primitive actions is a good thing, as this allows the agent to have refined control on the SUT, though the trade off is that it needs to put more effort in planning.

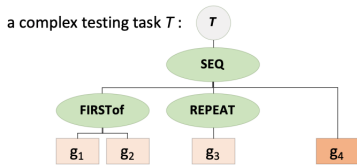
To do anything, the test agent must be given a *testing task*, see ① in Fig. 1, e.g. to verify an assertion as discussed above.

²<https://github.com/iv4xr-project/iv4xrDemo>

In terms of BDI agency, a task is a *goal*. When the agent manages to complete the goal, we say that it is ‘solved’. To solve a goal, a ‘tactic’ is needed. A tactic can be just a bunch of actions $\{\alpha_i\}$, each can be guarded with some reasoning logic g_i determining when it makes sense to execute the action. E.g. α_0 could be to interact with an in-game button, and its g_0 could require the agent to be physically close enough to the button (else the game might not allow the interaction); α_1 could be moving in the direction of the button, and its g_1 could require that the agent has a clear line of sight to the button.

Being a reactive program, the execution model of an agent is *very different* than e.g. a procedure or a service. An agent runs in (rapid) cycles. A cycle is either triggered by a clock tick or by an incoming event. At each cycle, the agent checks which actions in its current tactic are ‘enabled’; that is, having their logic-guard evaluates to true. One is then chosen, randomly, or according to some *policy*. The cycles are repeated until the goal is solved (or the agent runs out of budget). Note that the actions’ guards essentially form the reasoning part for solving the goal. Iv4XR additionally provides *combinators* similar to the idea of tactic combinators in theorem proving [13], [14] to structurally combine simpler tactics to construct more complicated ones. Indeed, programming a tactic might be non-trivial for testers, so this part should be made transparent, or at least mitigated. We will return to this later.

Some testing tasks can be expected to be non-trivial. To make them feasible for the test agent to solve, we can introduce subgoals. Goal-combinators can be used to specify how to combine them. The idea is similar to Behavior Tree from AI [15]; an example is visualized below:



E.g. the SEQ combinator formulates a task where all its subgoals must be solved in the specified sequence. Above, the actual testing task is g_4 , e.g. as in a previous example, to verify that a key room in a game is reachable. The subgoals $g_1..g_3$ can be thought as ‘lemmas’ to help in solving g_4 .

Figure 2 shows an example, at the code-level, of how a testing task is specified, assigned to an agent, and how to inspect the result of the task.

Many tasks are actually just variations of each other. E.g. a testing task to check the state of some in-game button b_0 is just a variation of a similar task for checking another button b_2 . We can therefore provide a *parameterized goal* to do this task, and instantiate it on demand whenever we need to verify some concrete button.

In Section II-A we mentioned ‘strategies’, and also as ② in Fig. 1. A strategy is a parameterized goal capturing a common subtask. When formulating testing tasks, we can imagine that testers have access to a library of these strategies; all they need to do is to instantiate them, and to arrange them towards

solving the testing task they have in mind.

Of course, someone needs to provide the strategies in the first place. We expect them to be quite SUT specific. E.g. strategies for a shooter game cannot be expected to be reusable for a train simulator. However, constructing them is a one-off investment, after which testers can keep reusing them to write automated testing tasks.

Since a strategy is essentially a goal, we need a tactic to solve it. So, the one-off investment also involves the development of common tactics. The basic building blocks for tactics are the primitive actions as provided by the SUT itself. To combine them, we indeed have the aforementioned tactic combinators, but additionally we also have access to a number of AI capabilities, indicated in ③ in Fig. 1:

- *Reasoning*: through actions’ guards as discussed before.
- *Navigation*: steering a virtual character to navigate through a 3D virtual world is not trivial since it typically also has to respect some physical laws. E.g. the virtual character would not be able to see nor walk through a solid obstacle; this complicates navigation a lot. Iv4XR implements the path finding algorithm A* so that a tactic can automatically steer a test agent to a given target location. When the agent want to inspects some virtual entity, but its location it not known (the tester may deliberately abstract away the location, to make the test more robust), the agent will then have to search the entity first. Iv4XR also implements an algorithm inspired by robot exploration to do this; for more on this see [16].
- *Learning*: Recall that an agent uses a ‘policy’ when determining which action among the set of enabled actions is to choose for execution. The default policy is just random, but a custom policy can be set, which in turn can be obtained through learning.

Technically, a goal is a predicate ϕ evaluated over a candidate C proposed by the tactic T_ϕ associated to ϕ . The goal is solved when T_ϕ finds a C such that $\phi(C)$ is true. To support unsupervised learning, ϕ can also be formulated as a cost function: when a tactic proposes a wrong C , $\phi(C)$ expresses an estimation of the remaining effort, starting from what we know about C , to find a solution. This would allow e.g. Reinforcement Learning to be deployed to learn a policy.

Note that a policy is used on *enabled* actions, which means it is used *in combination* with the agent’s own reasoning (through action guards) as the latter determines which actions are enabled.

III. INTERFACE PATTERN AND WORLD MODEL

To do its work the test agent will need an interface that lets it control the SUT and inspect its state. For XR systems, this is more challenging than e.g. browser-based applications. There is a large variation in the used UI technologies and there is no clear winner. Hence there is no common interfacing technology either. To make iv4XR usable for as much users as possible, the framework is provided open source and designed from the outset with extensibility in mind. The trade off is

```

1  var ttask = SEQ(
2    FIRSTof(g1, g2),
3    REPEAT(g3),
4    // g4:
5    assert_(agent,  $\beta \rightarrow \beta.wom.getElement(key) \neq null$ )
6
7  agent.attachState      (new W3DAgentState.java ())
8    .attachEnvironment(new EnvSUT())
9    .setDataCollector  (new TestDataCollector())
10   .setGoal            (ttask)
11   .budget             (1000)
12   .useDeliberation   (policy)
13
14  while(ttask.getStatus()==INPROGRESS) agent.update();
15
16  var verdicts = agent.getTestDataCollector();
17  assertTrue(verdicts.getNumberOfFailVerdictsSeen() == 0)

```

Fig. 2. An sample iv4XR code. Lines 1..5 specify a testing task. The last goal, g_4 , is a task to verify if an in-game entity key is present in the game; β represents the agent’s belief. Other goals are intermediate goals intended to guide the agent in solving g_4 . Line 11 assigns the aforementioned task to a test agent. Line 12 sets a computation budget for the agent, and line 13 sets an action-selection ‘policy’. Line 15 runs the agent in a simple loop. Finally, line 18 checks if the agent found no violation.

that a company using iv4XR first has to put some effort to instantiate its interface scheme, depicted in Figure 3.

Every test agent maintains a state (W3DAgentState), which includes what it believes to be the current SUT state. This is represented by a so-called World Object Model (WOM), or simply World Model. The agent’s state also contains a pointer to an ‘Environment’ (W3DEnvironment), which would provide a set of primitive actions used by the test agent to control or observe the SUT, such as to move some small distance, or to interact with an entity. The implementation of these actions is, however, SUT dependent, hence cannot be provided by W3DEnvironment itself. Developers therefore need to provide the implementation in the form of the class MyEnvironment that extends W3DEnvironment.

Each action from the Environment will also return a WOM, representing what the agent observes at the end of the action. Being an XR system, the SUT is assumed to represent a 3D world. A WOM represents a fragment of this world. E.g. it might list relevant entities in the world, along with their key properties. While the WOM returned by an action represents what the agent *currently* sees, the WOM maintained in the agent state *aggregates* all the received WOMs. While this maximizes the information the agent memorizes, some of the information in its WOM might not reflect the actual SUT state. The information is timestamped, but it is up to the agent to decide what to do with it.

Another important aspect of the WOM is that it represents a virtual world *structurally*, rather than visually. This allows agents to reason about the world much more accurately, which in turn also makes testing robust against all sorts of visual changes during the development; something which game designers can be expected to do quite often. Figure 4 shows the structure of a WOM. It is actually inspired by Domain Object Model (DOM) that is used by browsers to provide a common and structural interface to programmatically access

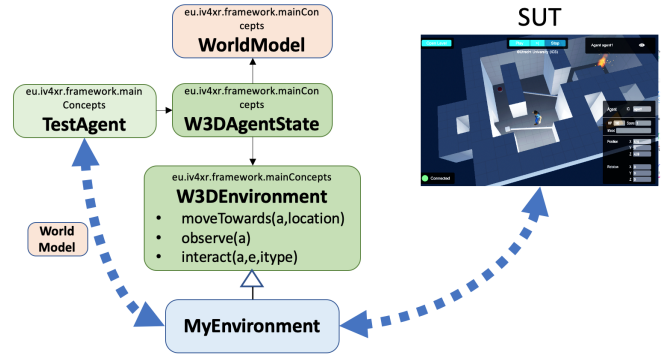


Fig. 3. The structure of the interface between iv4XR and the SUT. It is a variation of the Proxy Design Pattern [17] with W3DEnvironment and MyEnvironment taking the role of the abstract and concrete proxies.

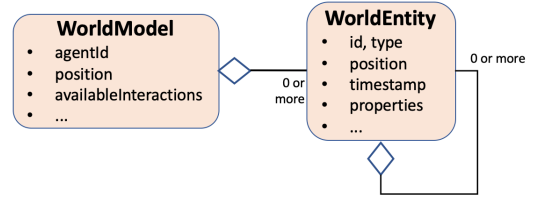


Fig. 4. The tree structure of the World Object Model (WOM). A world has 0 or more entities; each in turn may consist of multiple sub-entities. Since a WOM represents a 3D world, it keeps track e.g. the agent’s last known position in the world, as well as that of the entities.

and manipulate web pages. Importantly, a DOM is agnostic towards the actual ontology of the web page it represents. Similarly, a WOM is agnostic towards the game’s specific ontology, hence allowing test strategies to be crafted more generally.

IV. RELATED WORK

The most studied type of AI to aid automated testing is probably Reinforcement Learning (RL). One of the early works is that of Mariani et al. [18] where RL is used to help learning the behavioral model of the SUT, from which test cases are then generated. RL is mainly used to train a policy that optimize the discovery of new states (in other words, to optimize the test coverage). Later works, e.g. as in the use of RL in the GUI testing tool TESTAR [19], and similarly in approaches for mobile app testing [20], [21] still follow the same idea of how RL is exploited (so, to optimize coverage). In contrast, iv4XR sees AI mainly as an instrument for solving testing tasks.

Neural Networks (NN) have been proposed to be used to as artificial specifications [22], [23], [24]. The idea is to train an NN to approximate the behavior of a program $P(x)$, after which we can then use the NN as an oracle when testing P , in particular when the used testing algorithm generates a large amount of test cases (e.g. as in random testing). The approach suffers from false positives; even if there are only e.g. 5%, each one will have to be manually investigated. The issue is probably not to be blamed to the NN; it is a common issue in specification learning.

Iv4XR is not the first attempt to use agents to aid testing. Earlier works we can mention are [25], [26], [27]. These work went as far as using agents, and even BDI, but did not explore how they can actually exploit agent-based AI.

V. CONCLUSION, CURRENT STATUS, AND FUTURE WORK

We have given an overview of the iv4XR Framework. The Framework provides an agent-based approach to automatically solve testing tasks on interactive systems. It relies on agent-based AI, with possibilities to be combined with learning-based AI. The iv4XR Framework is under development and has currently reached a prototype level. Currently it is being piloted for testing 3D games, though as future work we would like to cover VR systems as well. Actual integration of learning AI is also future work, as well as the evaluation of iv4XR on more pilots.

REFERENCES

- [1] M. Wooldridge and N. R. Jennings, "Intelligent agents: Theory and practice," *The knowledge engineering review*, vol. 10, no. 2, 1995.
- [2] J.-J. C. Meyer, "Agent technology," in *Encyclopedia of Computer Science and Engineering*, B. W. Wah, Ed. John Wiley & Sons, 2008.
- [3] S. Russell and P. Norvig, *Artificial intelligence: a modern approach*. Prentice Hall, 2003.
- [4] A. Herzig, E. Lorini, L. Perrussel, and Z. Xiao, "BDI logics for BDI architectures: old problems, new perspectives," *KI-Künstliche Intelligenz*, vol. 31, no. 1, 2017.
- [5] R. H. Bordini, J. F. Hübner, and M. Wooldridge, *Programming multi-agent systems in AgentSpeak using Jason*. John Wiley & Sons, 2007, vol. 8.
- [6] M. Dastani, "2APL: a practical agent programming language," *Autonomous agents and multi-agent systems*, vol. 16, no. 3, 2008.
- [7] I. Prasetya, M. Dastani, R. Prada, T. E. Vos, F. Dignum, and F. Kifetew, "Aplib: Tactical agents for testing computer games," in *8th International Workshop on Engineering Multi-Agent Systems (EMAS)*, 2020.
- [8] T. E. Vos, P. M. Kruse, N. Condori-Fernández, S. Bauersfeld, and J. Wegener, "Testar: Tool support for test automation at the user interface level," *International Journal of Information System Modeling and Design (IJISMD)*, vol. 6, no. 3, pp. 46–83, 2015.
- [9] K. Claessen and J. Hughes, "QuickCheck: a lightweight tool for random testing of haskell programs," *ACM SIGPLAN notices*, vol. 46, no. 4, 2011.
- [10] I. S. Prasetya, "Budget-aware random testing with T3: benchmarking at the SBST2016 testing tool contest," in *IEEE/ACM 9th International Workshop on Search-Based Software Testing (SBST)*. IEEE, 2016.
- [11] G. Fraser and A. Arcuri, "EvoSuite: automatic test suite generation for object-oriented software," in *19th ACM SIGSOFT symposium and the 13th European conference on foundations of software engineering*. ACM, 2011.
- [12] C. Baier and J.-P. Katoen, *Principles of model checking*. MIT press, 2008.
- [13] D. Delahaye, "A tactic language for the system coq," in *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 2000.
- [14] M. J. Gordon and T. F. Melham, *Introduction to HOL A theorem proving environment for higher order logic*. Cambridge Univ. Press, 1993.
- [15] I. Millington and J. Funge, *Artificial intelligence for games*. Elsevier, 2009.
- [16] I. S. W. B. Prasetya, M. Voshol, T. Tanis, A. Smits, B. Smit, J. v. Mourik, M. Klunder, F. Hoogmoed, S. Hinlopen, A. v. 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, pp. 3–9.
- [17] E. Gamma, R. Helm, R. Johnson, J. Vlissides, and D. Patterns, "Elements of reusable object-oriented software," *Reading: Addison-Wesley*, 1995.
- [18] L. Mariani, M. Pezzè, O. Riganelli, and M. Santoro, "Automatic testing of gui-based applications," *Software Testing, Verification and Reliability*, vol. 24, no. 5, pp. 341–366, 2014.
- [19] S. Bauersfeld and T. E. Vos, "User interface level testing with testar; what about more sophisticated action specification and selection?" in *SATToSE*, 2014, pp. 60–78.
- [20] D. Adamo, M. K. Khan, S. Koppula, and R. Bryce, "Reinforcement learning for android gui testing," in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, 2018, pp. 2–8.
- [21] T. A. T. Vuong and S. Takada, "A reinforcement learning based approach to automated testing of android applications," in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, 2018, pp. 31–37.
- [22] M. Vanmali, M. Last, and A. Kandel, "Using a neural network in the software testing process," *International Journal of Intelligent Systems*, vol. 17, no. 1, pp. 45–62, 2002.
- [23] K. Aggarwal, Y. Singh, A. Kaur, and O. Sangwan, "A neural net based approach to test oracle," *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 3, pp. 1–6, 2004.
- [24] W. Prasetya and M. A. Tran, "Neural networks as artificial specifications, revisited," *EasyChair, Tech. Rep.*, 2019.
- [25] Y. Qi, D. Kung, and E. Wong, "An agent-based testing approach for web applications," in *29th Int. Computer Software and Applications Conference (COMPSAC)*, vol. 2. IEEE, 2005.
- [26] S. Paydar and M. Kahani, "An agent-based framework for automated testing of web-based systems," *Journal of Software Engineering and Applications*, vol. 4, no. 02, 2011.
- [27] X. Bai, B. Chen, B. Ma, and Y. Gong, "Design of intelligent agents for collaborative testing of service-based systems," in *6th Int. Workshop on Automation of Software Test*. ACM, 2011.