# AI Behavior Graphs: A Visual Toolkit for Defining NPC Specifications for Regression Testing

Pablo Gutiérrez-Sánchez[1,*], Marco A. Gómez-Martín[1,*], Pedro A. González-Calero[1,*] and Pedro P. Gómez-Martín[1,*]

[1]*Complutense University of Madrid, Madrid, Spain*

### Abstract

Reinforcement learning (RL) offers a promising approach for developing autonomous agents in various domains, including the creation of in-game characters. However, crafting these agents, and particularly designing reward functions for sequential decision-making, remains a significant challenge, often involving iterative trial-and-error processes until achieving satisfactory results. Consequently, these strategies often elude game designers and quality control teams, who could otherwise use them to automate testing procedures. This paper extends our prior work by introducing "AI Behavior Graphs," a visual toolkit designed to simplify the creation of behavior specifications for NPCs (Non-Player Characters). Our approach provides an intuitive graphical interface that enables designers to express their expectations for player-NPC interactions within a game level. These specifications are automatically translated into both Linear Temporal Logic (LTL) and Rabin automata, which can in turn be leveraged to dynamically generate reward functions during agent training. This not only expedites NPC development but also makes RL-based methodologies more accessible to a broader audience of game designers and quality assurance teams. Furthermore, it underscores a critical aspect of our approach: the ability to utilize these agents for playtesting game levels. This application ensures continuous validation of designer expectations throughout the development cycle, enhancing the overall game design process.

### Keywords

visual toolkit, automated game testing, game-playing AI, temporal logics, reinforcement learning

## 1. Introduction

In our previous work [1], we tackled the challenge of automating the generation of non-player characters (NPCs) capable of executing complex sequential tasks within video games. This effort was motivated by the need to enhance the efficiency of the testing and quality assurance (QA) process in game development, which has traditionally relied on human testers manually replaying level scenarios to identify and report bugs. These manual walkthroughs often follow a logical sequence of actions, representing possible ways for players to solve a level. In our approach, we represented these sequences as sets of steps—for example, in natural language something such as "Move to a platform, pick up an object, advance to the exit, all while avoiding

*II Congreso Español de Videojuegos, November 09–10, 2023, Madrid, Spain*

[*] Corresponding author.

pabgut02@ucm.es (P. Gutiérrez-Sánchez); marcoa@fdi.ucm.es (M. A. Gómez-Martín); pagoncal@ucm.es (P. A. González-Calero); pedrop@fdi.ucm.es (P. P. Gómez-Martín)

0000-0002-6702-5726 (P. Gutiérrez-Sánchez); 0000-0002-5186-1164 (M. A. Gómez-Martín); 0000-0002-9151-5573 (P. A. González-Calero); 0000-0002-3855-7344 (P. P. Gómez-Martín) enemy attacks, and ensure this sequence can be completed with the player losing less than half of their health points", providing a blueprint for player progression within the game environment.

Our objective was to equip NPCs with the capability to execute these logical sequences autonomously, enabling them to test game levels even in the presence of evolving game content or alterations to level logic—a common occurrence in game development.

While static or deterministic game environments allow for the use of pre-recorded human tester traces [2], the sensitivity of such traces to environmental changes or modifications poses clear limitations. These alterations can be subtle, such as minor adjustments to level platform placement, and yet potentially trigger incorrect outcomes in the execution traces, despite negligible impact on overall playability.

On the other hand, these changes are not confined to just the game environment—they can also affect the NPCs themselves. For instance, all enemies of a certain type may share traits like speed and strength, as well as preset behaviors. However, during the development of the game, tweaking these behaviors to fit the gameplay of specific levels can unintentionally make other levels harder or easier. This highlights the need for robust and adaptable mechanisms that can handle subtle shifts in the environment when conducting regression tests.

To address these challenges, prior research has explored the application of artificial intelligence (AI) techniques, including reinforcement [3] and imitation learning [4], or even blends of these strategies incorporating control structures like behavior trees (BTs) [5], to generate autonomous players for quality control testing. Although promising, these approaches often face overfitting issues or lack accessibility to non-technical project members like game designers and QA teams.

In our previous research, we described an approach that leveraged reinforcement learning (RL) guided by formal task specification language descriptions (Linear Temporal Logic, LTL [6]) to develop autonomous agents for testing. We provided a 3D experimentation environment within Unity3D [7], equipped with elements from stealth games, and integrated it with open-source machine learning libraries, notably ML-Agents [8]. However, our focus was solely on generating test agents from design specifications, laying the groundwork for future automation and streamlining of testing and training procedures.

Building upon this foundation, this paper introduces "AI Behavior Graphs," a visual toolkit that simplifies the creation of behavior specifications for NPCs, offering an intuitive graphical interface enabling designers to conveniently articulate their expectations for player-NPC interactions within a game level. These specifications are then automatically translated into both LTL and Rabin automata [9], which is a necessary step for the dynamic generation of reward functions during agent training.

This contribution streamlines NPC development and democratizes RL-based methodologies, making them accessible to a wider audience encompassing game designers and quality assurance teams. Additionally, our approach emphasizes the capability of utilizing these AI agents for playtesting game levels, ensuring ongoing validation of designer expectations throughout the development cycle. In doing so, it enriches the game design process and offers a promising path toward comprehensive automation of testing procedures in the gaming industry.

The remainder of the paper follows this structure: Section 2 details the proposed approach enabling designers to articulate potential solutions for levels, while Section 3 provides a high-level overview of the formal methodologies employed in this study. Section 4 presents a comprehensive description of the toolkit and its graphical editor, complete with an illustrative example showcasing the workflow of a designer compiling a level specification. Lastly, we conclude with our final remarks and outline future research directions.

## 2. Temporal Logic Specifications for Regression Testing

In this section, we outline the perspective of the designer, specifying the particular information and format that designers should provide us with to enable us to effectively train an agent to emulate the player and navigate the level according to the designer's intended solutions.

### 2.1. Context

When game designers create a video game level, they usually have a particular solution in mind for how they expect players to complete it. For instance, in an adventure game level:

1. The player starts in a dark dungeon and must find a torch to light their way and exit this first chamber.
2. After exiting the chamber, they encounter an area being patrolled by a set of enemies. This area contains a hidden switch that needs to be activated in order for the player to unlock the level door.
3. The player must figure out a way to approach the switch without being detected by the enemies and activate it.
4. Lastly, they may access treasures, and exit the level.

It's important to note that these specifications use a high-level and non-specific vocabulary, allowing for multiple ways to satisfy them. For instance, how the player defeats the level's enemies guarding the doors is irrelevant as long as they deal with all of them.

Once a plan for completing the level is established, it can be used to create a quality assurance test within a test suite. Testers can then follow these steps to verify if the level can be completed as intended. These checks fall under the category of "regression testing": sets of tests ideally conducted regularly to ensure that changes made during development do not disrupt the specification and that it remains feasible to meet the designer's requirements.

However, as development progresses and new content is added, maintaining comprehensive coverage of all regression tests becomes impractical. This is where automatic testing methodologies come into play to streamline the process and enhance product coverage and quality control. From a designer's perspective, the goal is to specify planned level solutions in a standardized way and generate periodic reports to check compliance at any given point in time.

### 2.2. Original Methodology

The tools developed in this paper build upon our previous work's initial workflow. In this workflow, designers can specify how a level's resolution process should unfold using pseudo-natural language combined with logical operators. This specification then serves as the basis for automatically generating an NPC (Non-Player Character) tasked with fulfilling, to the best of its ability, the requirements outlined in the task description.

Behind the scenes, this automated NPC generation process leverages techniques from the realm of deep reinforcement learning, outlined in Section 3. This approach adds a layer of abstraction, effectively eliminating the requirement for expertise in statistical learning when training a bot to meet the specified criteria. This development holds significant relevance not just for designers but also for machine learning practitioners in the field of video games. It addresses the issue of crafting intricate and often opaque reward functions, which implicitly dictate the behavior of the agent—a problem commonly known as "reward engineering." Instead, our approach automatically generates these reward functions using clear expressions based on domain knowledge provided by experts in the field.

It is important to underscore that our primary focus is not on creating agents that learn tasks entirely from scratch. Rather, we concentrate on translating a designer's preconceived plan of action into a trainable system that can learn to execute it with minimal friction between the designer's intent and the learning process. Additionally, when discussing pseudo-natural language here, we are referring to specifications based on formal logic rather than plain text with such content. Nevertheless, we find it helpful to provide a verbal explanation of the specification to assist readers in understanding the design intention behind it.

## 3. Temporal Logics for Reinforcement Learning

In recent times, particularly in the field of robotics, there has been a growing interest in using temporal logics (TL) to define reward functions in reinforcement learning (RL) algorithms [10]. Among these logics, those enabling quantitative semantics, termed robustness [11], have gained significant attention, as they quantify how well a state trace adheres to a specification. This robustness metric can serve as a natural reward function, facilitating the definition of sequential tasks, sub-objectives, temporal constraints, and behavioral restrictions.

Signal Temporal Logic (STL)[12] is a widely employed temporal logic language, well-suited for robot control tasks, but its reward generation process applies only to complete trajectories, resulting in sparse rewards and complicating the learning of lengthy or intricate tasks. Alternatively, Truncated Linear Temporal Logic (TLTL)[6] allows formula evaluation over finite trajectories of varying lengths. Early applications of TLTL in RL were limited to a single reward evaluation at the end of training episodes. Later work introduced Finite State Predicate Automata (FSPAs) to capture temporal dependencies and offer dense rewards based on robustness variations between transitions [13], making it a more suitable tool for complex task learning.

We aim to select a logic that combines qualitative (True or False predicates) and quantitative semantics (continuous measure of formula satisfaction). TLTL fulfills these criteria, allowing the incorporation of complex intentions, domain knowledge, and constraints into task specifications.

TLTL formulas are defined over predicates of the form $f(s) < c$, where $f : \mathbb{R}^n \to \mathbb{R}$ and $c$ is a constant, and include operators like $\Diamond$ (eventually), $\Box$ (always), $\mathcal{U}$ (until), and $\bigcirc$ (next), as well as boolean operators ($\wedge,\ \vee,\ \Rightarrow$), with the following syntax:

$$\phi := \top \mid f(s) < c \mid \neg\phi \mid \phi \wedge \psi \mid \phi \vee \psi \mid \Diamond\phi \mid \Box\phi \mid \phi\,\mathcal{U}\,\psi \mid \phi\,\mathcal{T}\,\psi \mid \bigcirc\phi \mid \phi \Rightarrow \psi \qquad (1)$$

As an example of a TLTL formula, the middle part of the specification in Section 2.1, requiring the player to eventually approach a switch and next activate it at some point, while ensuring that no enemy ever gets too close to the character can be expressed in TLTL as:

$$\phi := \Diamond\, (\texttt{switchClose} \wedge \bigcirc\, (\Diamond\, \texttt{switchOn})) \wedge \Box\, \neg\texttt{enemyClose} \qquad (2)$$

While the formula representation of this predicate might seem concise, understanding the logic behind the temporal operators may not be immediately clear upon initial exposure to these languages. In this example, the $\wedge$ operator is used to connect the two fundamental parts of the specification that must be satisfied by the NPC, which will be clarified below.

The first sub-predicate, $\Diamond\, (\texttt{switchClose} \wedge \bigcirc\, (\Diamond\, \texttt{switchOn}))$, corresponds to the notion that the player must find the switch and then activate it. From a design perspective, this is a requirement that must be achieved at some point in the game to fulfill the specification, which is why this predicate begins with a $\Diamond$ operator, indicating the need for the associated predicate to be fulfilled at some future point. A slightly more subtle aspect is the use of the $\bigcirc$ operator to represent that after finding the switch, it must be activated as a next step. In practice, the $\bigcirc$ operator requires the associated condition to be met in the following simulation time step (e.g. if our predicate were solely $\bigcirc\, \phi$ we would expect $\phi$ to be true in the second time step of the run). This is why it is common to connect it with a $\Diamond$ operator to ensure that the specified condition is met at some later time. This allows us to define sequences of conditions that must be eventually met one after another by simply adding blocks of the form $\phi_A \wedge \bigcirc\, (\Diamond\, \phi_B)$.

The second sub-predicate, $\Box\, \neg\texttt{enemyClose}$, corresponds to the idea that no enemy should get too close to the player. From a logical standpoint, this is a global constraint that is always in effect and can be modeled with a $\Box$ operator, followed by what we want to always hold true: that no enemy is within a certain safe distance from the player. $\Box$ operators are particularly useful for modeling level constraints like always having a certain item equipped or never allowing health points to drop below a certain threshold.

Going back to TLTL's quantitative semantics, or how well a sequence of states adheres to a specification, the degree of robustness of a trajectory with respect to a specification $\phi$ is represented by $\rho(s_{t:t+k}, \phi)$, where $s_{t:t+k}$ denotes a state sequence. Positive values indicate adherence to $\phi$, while negative values signify non-adherence. Despite some limitations, the robustness definition proposed by [6] was used in our previous work due to its ease of implementation, but other feasible alternatives can be found in the literature [14].

The degree of robustness based on a specification $\phi$ can be employed with reinforcement learning, where it serves as a reward function. This reward function awards robustness at the end of an episode, aligning with natural language and simplifying the specification process. However, this results in sparse rewards, which hinders learning complex tasks.

To address this issue, [13] and [15] propose generating Finite State Predicate Automata (FSPAs) from TLTL predicates to capture sequences of actions needed to satisfy a task. FSPAs consist of transitions equipped with TLTL predicates, triggering when their robustness is positive and maximal among the set of currently positive transitions. This conversion, which is always possible to perform as shown by [16], allows step-based rewards rather than episodic ones based on the local variations in robustness for the current automaton state. While various reward allocation algorithms exist, we adopt an adaptation of the one introduced by [15] in which we opt to build the FSPAs by means of a structure called a Rabin automaton [9]; this is a finite state machine which makes its runs on infinite words and performs reasoning about them.

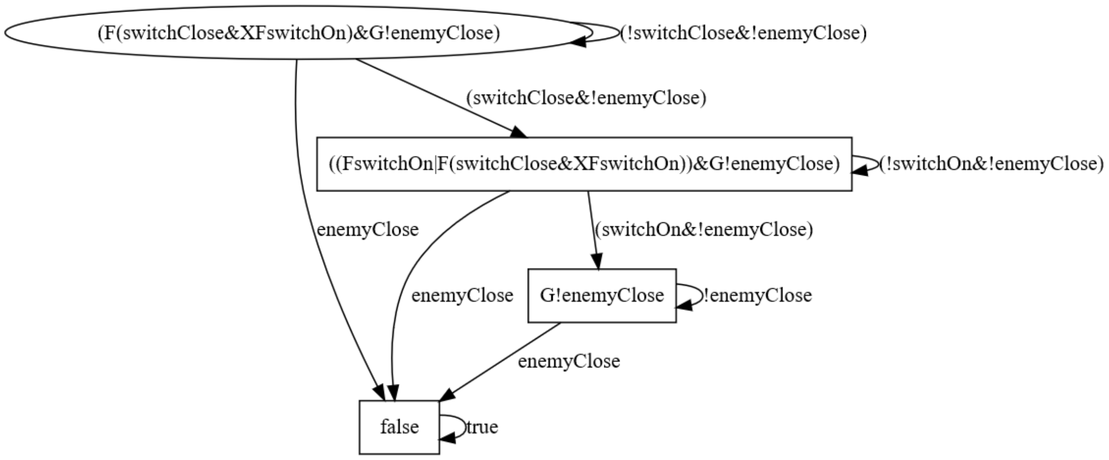

(F(switchClose&XFswitchOn)&G!enemyClose)
(!switchClose&!enemyClose)
(switchClose&!enemyClose)
((FswitchOn|F(switchClose&XFswitchOn))&G!enemyClose)
(!switchOn&!enemyClose)
enemyClose
(switchOn&!enemyClose)
enemyClose
G!enemyClose
!enemyClose
enemyClose
false
true

**Figure 1:** Rabin Automaton-based FSPA for the specification in Formula 2.

Continuing with the previous example, the FSPA shown in Figure 1 allows us to represent the specification from Formula 2 using 4 states and the transitions from the diagram. Please note that in this format, the initial state is represented by an oval and we have $F \equiv \Diamond$ (eventually), $G \equiv \Box$ (always), and $X \equiv \bigcirc$ (next). It is important to underscore that, although the automaton in the figure may be understandable with a little patience, the transitions and labels it generates are generally not trivial and highlight the structural complexity that can unexpectedly arise behind seemingly simple specifications.

## 4. The AI Behavior Graphs Toolkit

As highlighted in the previous section, while LTL is valuable for automatically generating reward functions tailored to human specifications, manually crafting these specifications can quickly become a cumbersome task. Hence, we introduce The Unity AI Behavior Graphs Toolkit, a dedicated tool engineered to simplify the creation and training of LTL-driven NPC behaviors in Unity 3D. This toolkit empowers designers to outline NPC behaviors through an intuitive visual node editor and subsequently generate reward functions for training reinforcement learning agents, streamlining the process described in Section 3.

### 4.1. Creating new Graph Specifications

Once the tool has been added to the Unity project, the designer can start interacting with the graphical behavior window from the engine editor through a dedicated inspector menu option. This opens a window like the one shown in Figure 2, with an empty default graph that includes a blackboard with no variables and the predicate return node. In the graph view, two fundamental elements are present:

- **Blackboard**: Used for storing parameters that can be referenced by nodes in the graph. Parameters are categorized based on their type (`GameObject`, `Vector3`, `String`, etc.).

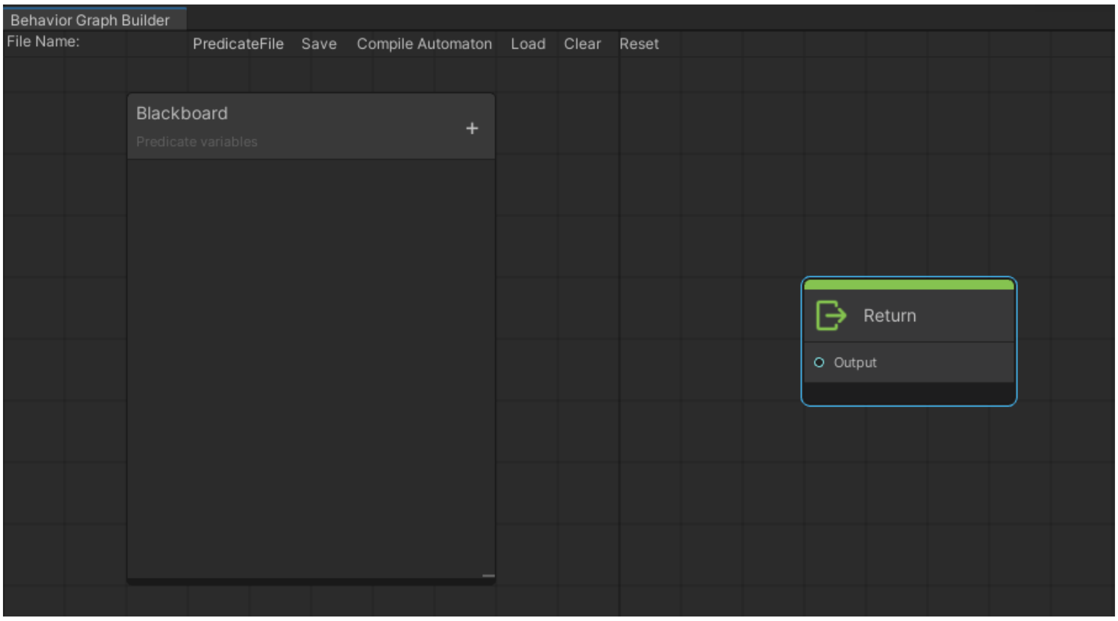


**Figure 2:** Default Editor Window.

- **Node Editor**: A graphical editor that facilitates the addition, deletion, editing, and management of connections between nodes. To introduce a new node to the graph, the user can right-click in the editor and select the "Create Node" option to access the node exploration panel. This explorer is structured into nested categories, including `Flow` (for core temporal logic operators) and `Conditions` (for basic condition operators). It is also possible to include new categories and nodes, as detailed in Section 4.2.

The input parameters of a node can be other nodes or predicates in the graph, in which case they are represented by input ports to receive a connection through an edge, or blackboard variables. In the latter case, the parameter will be listed in a dedicated section at the bottom of the node along with a drop-down selector and a button with the + symbol. Through the drop-down, the user can select the blackboard variable that will be linked at run-time with the input, while the button allows creating a new variable of the appropriate type on the blackboard.

It is important to note here that while there can be multiple disconnected groupings of nodes in the behavior graph, the only component that will be considered at run-time is the one linked with the special return node (green in color, always one unique node of this type per graph).

As an example of a graph representation that can be created with this tool, Figure 3 depicts a behavior graph along with its blackboard for the specification from Formula 2 that was explained in Section 3. Note that the original predicates have been expressed here by means of more general nodes (e.g. `switchClose` becomes `targetClose`). This sort of tree representation is often more intuitive than the original formula, and easier to understand for non-technical profiles.

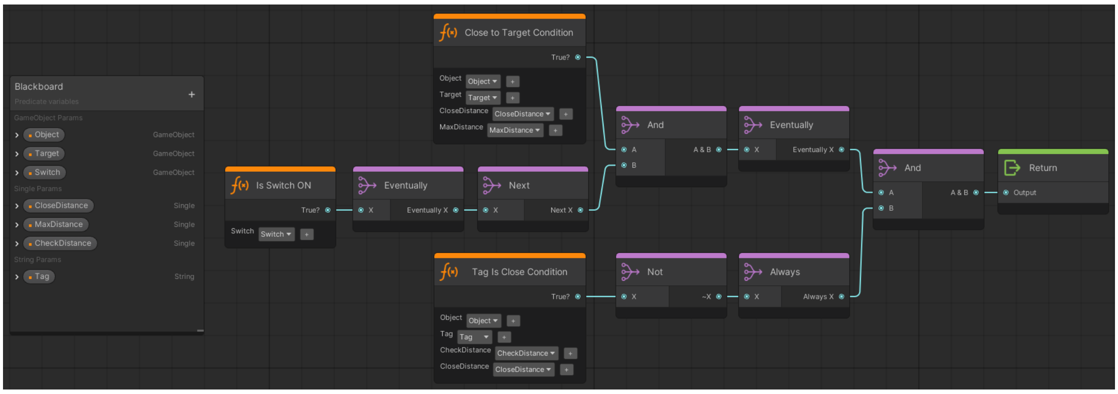


**Figure 3:** Example editor graph for the specification from Formula 2.

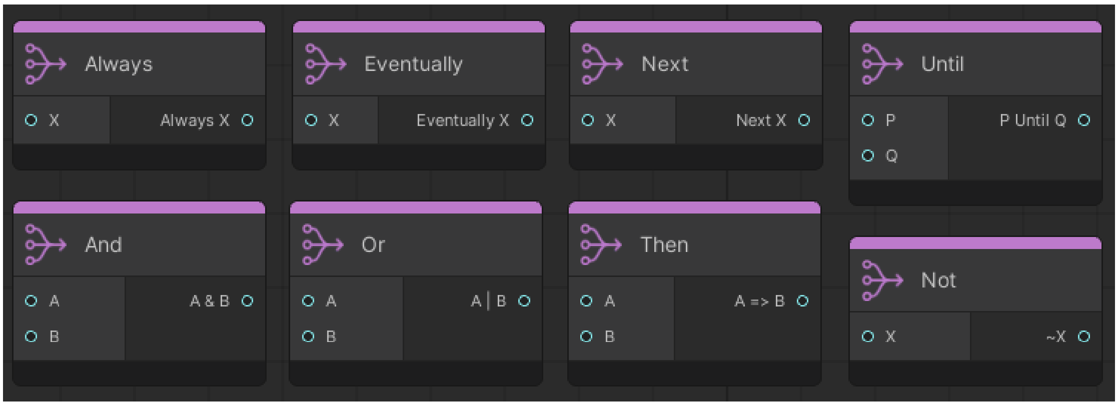


**Figure 4:** Currently available flow nodes.

## 4.2. Adding Custom Editor Nodes

The AI Behavior Graphs Toolkit differentiates between two types of nodes based on their role in the specification to be constructed:

- **Flow Nodes** correspond to standard operators in Linear Temporal Logic that take a set of LTL predicates as parameters (eventually, always, until, etc.). Note that this is a closed set of operators and that no other composite nodes are supported to date, meaning that user-created nodes are not allowed to introduce references to other predicates as parameters. A visual overview of the available flow nodes can be found in Figure 4.
- **Condition nodes** correspond to robustness functions in LTL and represent a condition on a set of parameters whose truth value is not given in a boolean format (`true` or `false`) but rather by a numerical value representing its robustness.

Although some default simple conditions are included in the project, it is possible to manually define new condition nodes that can be subsequently included in behavior graphs. To do this, the user must extend the abstract class `TLTLPredicate`, as shown in Figure 5 for an example

```
[Predicate("Conditions/Less Than", "A < B")]
public class LessThanPredicate : TLTLPredicate {
    [InParam("A")]
    public float A { get; set; }
    [InParam("B")]
    public float B { get; set; }
    public override float EvaluateRobustness() {
        return B - A;
    }
}
```

**Figure 5:** Defining a custom node for $A < B$ predicate.

of a predicate that computes the robustness of $A < B$. The most important function here is `EvaluateRobustness`, which allows returning a numerical value to represent the extent to which the predicate is satisfied at the current moment. While there are no upper or lower bounds for the value returned by the function, it is recommended to keep it within a normalized range of $[-1, 1]$ in order to ensure that interactions with other operators and conditions are consistent and to prevent it from being overwhelmed by the magnitude of nearby predicates.

On the other hand, since a predicate of this type is nothing more than a function that returns a value based on inputs, it is also important to mention these parameters. The input parameters that will be available in the predicate's robustness calculation must be declared in the class that extends `TLTLPredicate` with the annotation `[InParam(“paramName”)]`, specifying the name by which the user wishes to refer to the parameter from the associated node in the graphical editor. Note that this name does not have to match the variable name in the class.

Finally, for the predicate to be available as a node in the editing tool, it is essential that the implemented class is accompanied by the annotation `[Predicate(“path/to/condition”, “predicateOutputName”)]`. Remembering the structure of the node explorer in the graphical editor, each node can be included in a different category, potentially nested in a chain of subcategories, as in the case of `LessThanPredicate`, which appears under the `Conditions` category (see Figure 6). The first input parameter of the preceding annotation precisely corresponds to this “path” of categories and must be defined manually by the class implementer. On the other hand, the second parameter of the annotation corresponds to the text that will appear in the graphical editor to represent the node's output value, and can be used to further clarify the nature of the condition to the user.

## 4.3. Compiling Automata

Once a satisfactory graph for an NPC is ready, the next step is to compile it into an automaton for run-time use. In reality, what is generated during the behavior editing process through the visual tool is just a specification in LTL and the metadata for its graph representation, neither of which can be directly used to reward an agent during training. To make it usable in that context, we need to convert the original specification into a structure that allows local reasoning about the predicate, abstracting all dependencies and temporal operators into states where we

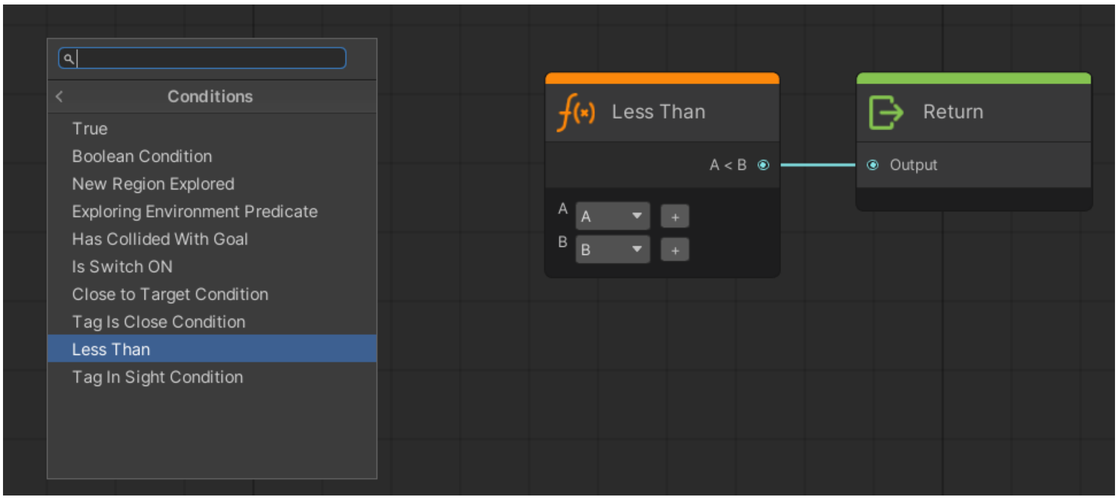
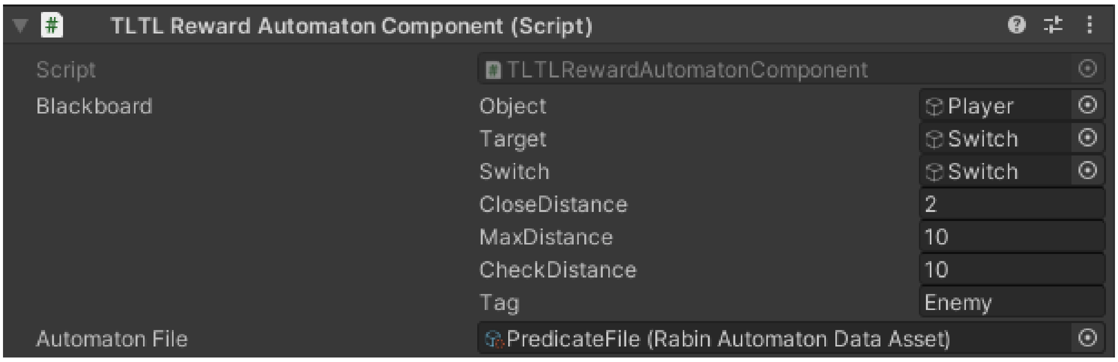


**Figure 6:** Less Than Node.



TLTL Reward Automaton Component (Script)

|  |  |  |
|---|---|---|
| Script | TLTLRewardAutomatonComponent | |
| Blackboard | Object | Player |
| | Target | Switch |
| | Switch | Switch |
| | CloseDistance | 2 |
| | MaxDistance | 10 |
| | CheckDistance | 10 |
| | Tag | Enemy |
| Automaton File | PredicateFile (Rabin Automaton Data Asset) | |

**Figure 7:** Reward Automaton Component as seen in the Unity Editor.

know our current position in the process. This can be achieved through the use of the FSPAs introduced in Section 3, in our case constructed by means of Rabin Automata.

However, the conversion of a specification written in Linear Temporal Logic into a Rabin automaton poses a complex challenge. To facilitate it, we currently employ an external open-source Java library known as Rabinizer 3 [17], which is executed via a C# Process from the Unity Editor. When selecting the "Compile Automaton" option from the editor menu, the conversion process is triggered. This starts by transforming the current predicate into a format acceptable to the Rabinizer library. Once the output automata is ready, the process proceeds to parse the result into a Rabin Automaton Data Asset, which can be used to perform inference at run-time. Note that the generated asset contains only references to the necessary classes for instantiating different node types in the specification at run-time, as well as the blackboard variables they reference. These variables remain without actual values until assigned to an entity in the scene.

On a different note, the graph in Figure 3 does not accurately reflect the intention we want to convey in the specification. The predicates used are mostly generic (e.g., "object near the target" instead of “player near the switch”) and require specific entities to convey the designer’s real intent. To perform this assignment, the toolkit provides the `TLTLRewardAutomatonComponent`, which acts as a bridge between the automaton-format specification asset provided as a parameter, its blackboard variables, and the elements in the scene. This component is typically added to the NPC’s game object that should satisfy the specification. An example after assigning the asset compiled from Figure 3 with the necessary input parameters can be seen in Figure 7. Here, some parameters are literals, while others are direct references to game objects.

At run-time, this component proceeds to instantiate the automaton’s transitions with specific instances of the predicates to be executed during the game to evaluate conditions and user-introduced references, resulting in what we call a `TLTLRewardAutomaton`. This acts as a data container that does not update itself during execution, delegating this task to other components that can trigger such updates by calling the automaton’s `Tick` method. For further details, including possible ways to train an agent to adhere to the specification, please refer to [1].

## 5. Conclusions and Future Work

This paper introduced “AI Behavior Graphs,” a user-friendly toolkit that simplifies NPC development and RL methodologies, making them accessible to game designers and quality assurance teams. While we are confident in the toolkit’s capabilities, it is important to note that designers without prior experience in temporal logics may encounter a learning curve. To ensure the practical relevance and broader impact of our toolkit, we are committed to refining AI Behavior Graphs based on real-world feedback from industry professionals. Our goal is to not only make this tool useful for research but also a valuable asset in game development, aligning it with the preferences of game designers. By bridging the gap between research and practical game development, our aim is to create a more accessible and valuable resource for the industry, ultimately advancing the synergy between AI and game development.

## Acknowledgments

This work was supported by the Ministry of Science and Innovation (PID2021-123368OB-I00).

## References

[1] P. Gutiérrez-Sánchez, M. A. Gómez-Martín, P. A. González-Calero, P. P. Gómez-Martín, Reinforcement learning with temporal logic specifications for regression testing npcs in video games, in: 2023 IEEE Conference on Games (CoG), 2023 (forthcoming).

[2] M. Ostrowski, S. Aroudj, Automated Regression Testing within Video Game Development, GSTF Journal on Computing (JoC) 3 (2013) 10. URL: https://doi.org/10.7603/s40601-013-0010-4. doi:10.7603/s40601-013-0010-4.

[3] J. Bergdahl, C. Gordillo, K. Tollmar, L. Gisslen, Augmenting Automated Game Testing with Deep Reinforcement Learning, in: 2020 IEEE Conference on Games (CoG), IEEE, Osaka, Japan, 2020, pp. 600–603. URL: https://ieeexplore.ieee.org/document/9231552/. doi:10.1109/CoG47356.2020.9231552.

[4] S. Ariyurek, A. Betin-Can, E. Surer, Automated Video Game Testing Using Synthetic and Humanlike Agents, IEEE Transactions on Games 13 (2021) 50–67. URL: https://ieeexplore.ieee.org/document/8869824/. doi:10.1109/TG.2019.2947597.

[5] P. Gutiérrez-Sánchez, M. A. Gómez-Martín, P. A. González-Calero, P. P. Gómez-Martín, Reinforcement Learning Methods to Evaluate the Impact of AI Changes in Game Design, Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment 17 (2021) 10–17. URL: https://ojs.aaai.org/index.php/AIIDE/article/view/18885.

[6] X. Li, C.-I. Vasile, C. Belta, Reinforcement learning with temporal logic rewards, in: 2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), IEEE, Vancouver, BC, 2017, pp. 3834–3839. URL: http://ieeexplore.ieee.org/document/8206234/. doi:10.1109/IROS.2017.8206234.

[7] P. Gutiérrez-Sánchez, M. A. Gómez-Martín, P. A. González-Calero, P. P. Gómez-Martín, Liquid snake: a test environment for video game testing agents, in: R. Lara-Cabrera, A. J. F. Leiva (Eds.), Actas del I Congreso Español de Videojuegos, Madrid, Spain, December 1-2, 2022, volume 3305 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2022. URL: https://ceur-ws.org/Vol-3305/paper7.pdf.

[8] A. Juliani, V.-P. Berges, E. Teng, A. Cohen, J. Harper, C. Elion, C. Goy, Y. Gao, H. Henry, M. Mattar, D. Lange, Unity: A General Platform for Intelligent Agents, arXiv:1809.02627 [cs, stat] (2020). URL: http://arxiv.org/abs/1809.02627, arXiv: 1809.02627.

[9] B. Khoussainov, A. Nerode, Rabin Automata, Birkhäuser Boston, Boston, MA, 2001, pp. 249–328. URL: https://doi.org/10.1007/978-1-4612-0171-7_5. doi:10.1007/978-1-4612-0171-7_5.

[10] H.-C. Liao, A Survey of Reinforcement Learning with Temporal Logic Rewards (2020).

[11] V. Raman, A. Donzé, M. Maasoumy, R. M. Murray, A. Sangiovanni-Vincentelli, S. A. Seshia, Model Predictive Control for Signal Temporal Logic Specification, arXiv:1703.09563 [cs] (2017). URL: http://arxiv.org/abs/1703.09563, arXiv: 1703.09563.

[12] O. Maler, D. Nickovic, Monitoring Temporal Properties of Continuous Signals, in: Y. Lakhnech, S. Yovine (Eds.), Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems, Springer, Berlin, Heidelberg, 2004, pp. 152–166. doi:10.1007/978-3-540-30206-3_12.

[13] X. Li, Z. Serlin, G. Yang, C. Belta, A formal methods approach to interpretable reinforcement learning for robotic planning, Science Robotics 4 (2019) eaay6276. URL: https://www.science.org/doi/10.1126/scirobotics.aay6276. doi:10.1126/scirobotics.aay6276.

[14] N. Mehdipour, C.-I. Vasile, C. Belta, Specifying User Preferences Using Weighted Signal Temporal Logic, IEEE Control Systems Letters 5 (2021) 2006–2011. URL: https://ieeexplore.ieee.org/document/9309020/. doi:10.1109/LCSYS.2020.3047362.

[15] X. Zhao, M. Campos, Reinforcement Learning Agent Training with Goals for Real World Tasks, arXiv:2107.10390 [cs] (2021). URL: http://arxiv.org/abs/2107.10390.

[16] K. Y. Rozier, Explicit or Symbolic Translation of Linear Temporal Logic to Automata, Thesis, Rice University, 2013. URL: https://scholarship.rice.edu/handle/1911/71687.

[17] Z. Komárková, J. Křetínský, Rabinizer 3: Safraless translation of ltl to small deterministic automata, in: F. Cassez, J.-F. Raskin (Eds.), Automated Technology for Verification and Analysis, Springer International Publishing, Cham, 2014, pp. 235–241.