Assessing the Interpretability of Programmatic Policies with Large Language Models

Zahra Bashir, Michael Bowling, Levi H. S. Lelis
Department of Computing Science, University of Alberta, Canada
Alberta Machine Intelligence Institute(Amii)
zbashir1, mbowling, levi.lelis@ualberta.ca

Abstract

Although the synthesis of programs encoding policies often carries the promise of interpretability, systematic evaluations were never performed to assess the interpretability of these policies, likely because of the complexity of such an evaluation. In this paper, we introduce a novel metric that uses large-language models (LLM) to assess the interpretability of programmatic policies. For our metric, an LLM is given both a program and a description of its associated programming language. The LLM then formulates a natural language explanation of the program. This explanation is subsequently fed into a second LLM, which tries to reconstruct the program from the natural-language explanation. Our metric then measures the behavioral similarity between the reconstructed program and the original. We validate our approach with synthesized and human-crafted programmatic policies for playing a real-time strategy game, comparing the interpretability scores of these programmatic policies to obfuscated versions of the same programs. Our LLM-based interpretability score consistently ranks less interpretable programs lower and more interpretable ones higher. These findings suggest that our metric could serve as a reliable and inexpensive tool for evaluating the interpretability of programmatic policies.

1 Introduction

There is a growing interest in the use of programmatic representations of policies to solve sequential decision-making problems, both in single-agent [Verma *et al.*, 2018; Qiu and Zhu, 2022] and multi-agent settings [Mariño *et al.*, 2019; Medeiros *et al.*, 2022]. This interest is justified as one can provide strong inductive bias to the learning process through the domain-specific language defining the space of programs. This bias can allow programmatic policies to generalize more easily to unseen settings [Inala *et al.*, 2020] and make them more amenable to verification [Bastani *et al.*, 2018].

Previous work on programmatic policies also often emphasizes interpretability. However, systematic studies that assessed the interpretability of these policies were never performed. A common method is to present specific programs and claim their interpretability [Verma *et al.*, 2018; Aleixo and Lelis, 2023]. The scarcity of comprehensive evaluations could be attributed to the fact that such studies are time consuming and costly, mainly because they would involve human programmers. This lack of a thorough analysis hinders our understanding of what precisely makes a programmatic policy interpretable. For instance, neural networks can be viewed as programs written in a domain-specific language that allows the addition of layers and nodes to the neural architecture—clearly, the programmatic framing for policies does not guarantee interpretability. So, what are the properties that make a programmatic policy interpretable?

Any viable approach to addressing this question is likely to involve evaluating the interpretability of programmatic policies. In this paper, we introduce a simple and cost-effective methodology to assess program interpretability and demonstrate its application to programmatic policies. Our methodology uses large language models (LLMs) [Brown *et al.*, 2020] to assign an interpretability score to a program. We call this score the LLM-based INTerpretability (LINT) score. In our methodology, we use an instance of an LLM to generate a natural-language explanation of a program. This explanation is given as input to another instance of an LLM, which is asked to reconstruct the program described in the explanation. A third instance of an LLM verifies that the explanation is in natural language and does not provide step-by-step programming instructions on how to write the program. The LINT score is the value of a metric comparing the behavior of the original and reconstructed programs. We introduce general behavior metrics for sequential decision-making problems.

The evaluation of our methodology is based on methods from the program obfuscation literature [Collberg and Nagra, 2009]. Obfuscated programs are designed to be non-interpretable, and some obfuscation techniques allow us to construct programs with different levels of obfuscation. Assuming that obfuscation can be used as a proxy for interpretability, we hypothesize that the LINT scores negatively correlate with the degree of obfuscation we apply to the programs. Our methodology also includes the use of programmatic policies written by humans. Our premise is that since these policies are human-written, they should be inherently interpretable, and thus be scored as such in our metric.

Empirical results on classical programming problems and

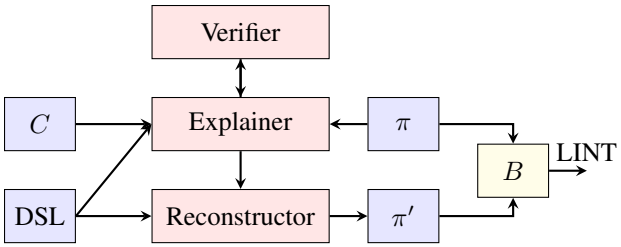

Figure 1: General overview of LINT. The Explainer receives a program π, a set of constraints C, and a description of the DSL in which π was written; it produces a natural language explanation of π, which is checked by the Verifier. The explanation is provided as input, along with the description of the DSL, to the Reconstructor, which attempts to reconstruct π from the explanation, thus producing π'. B scores the similarity (or dissimilarity) of π and π'.

programmatic policies for playing MicroRTS [Ontañón *et al.*, 2018] show that the LINT scores strongly and negatively correlate with the level of obfuscation of the programs evaluated. Although user studies should still be the gold standard for evaluating interpretability, our results suggest that LINT can be used as a reliable and inexpensive tool to help drive research in interpretable programmatic policies.

2 LINT: LLM-based Interpretability Score

We define the function $B(\pi_1, \pi_2)$ as a similarity metric for the behavior of two programs. We consider functions B that return a number between 0 and 1, where the value of 0 represents the most dissimilar behavior for the two programs and 1 represents identical behavior for the programs.[1] We denote by $\mathcal{L}_e(\pi, G, C)$ an LLM that receives a program π, a domain-specific language (DSL) G, a set of constraints C, and returns a natural language explanation of π. We refer to this LLM as **explainer**. We denote by $\mathcal{L}_r(e, G)$ an LLM that receives a natural language explanation e of a program and a DSL G, and returns a program accepted by the language G that exhibits the behavior described in e. We refer to this LLM as the **reconstructor**. Both the explainer and the reconstructor receive a natural language description of G with a context-free grammar that specifies the programs G accepts.

Given a set Π with n programs and a behavior metric B, the LINT score is computed as

$$\text{LINT}(\Pi, B, G, C) = \frac{1}{n} \sum_{\pi \in \Pi} B\Big(\pi, \mathcal{L}_r\big(\mathcal{L}_e(\pi, G, C), G\big)\Big). \tag{1}$$

The LINT score of set Π is the average value of how similar the programs in Π are from the reconstructed ones. We define the LINT score over a set of programs to measure the interpretability of the programs a system generates. However, in our experiments we also consider the case where $|\Pi| = 1$.

Figure 1 shows a schematic view of how the LINT score is computed for a program π.

[1] In our experiments, we also consider a dissimilarity metric, where 0 represents the most similar and 1 the most dissimilar behavior.

2.1 Set of Constraints for Explanation

The above formulation considers a set of constraints C to generate the explanation of a program. C prevents the LLM from generating the explanation of the program with non-interpretable elements that communicate the program to the other LLM. The constraints are instructions in the LLM prompt. We include the constraints shown in the list below.

1. *Try to understand what is happening in the code and explain it in natural language to someone who wants to learn about this program.*
2. *Write a high-level explanation and do not explain the code line-by-line, but it is fine to include numbers in your natural language explanation.*
3. *You must not use programming language jargon as people not familiar with programming might not understand the explanation.*

Without these constraints, the LLM could generate line-by-line instructions of how to reconstruct the program. For example, even if the program was an implementation of the neural network, the LLM could provide instructions on how to implement the architecture and copy the weights of the model. Although this explanation could allow the second LLM to reconstruct the program, the original program might not be interpretable. Even with these constraints, the LLM occasionally generates explanations that use programming instructions such as *"[...] after a nested for-loop [...]"*. We use a third LLM, the **verifier**, to partially check for the constraints. Specifically, we ask it to verify whether the explanation uses computer programming jargon and/or keywords of the DSL. If the verifier answers 'yes' to the use of jargons, then we sample another explanation from the explainer.

2.2 Multiple Trials

Due to the stochastic nature of how the LLMs generate the explanations and programs, we repeat k times the computation of $B(\pi, \mathcal{L}_r(\mathcal{L}_e(\pi, G, C), G))$ in Equation 1 and use in the summation the best B-value of the k trials. Trials are carried out by generating one explanation for each program, and each explanation is used to generate k programs. The value of k should be large enough to account for the variance of the LLM generation and small enough to prevent the LLM from reconstructing the original program by chance. Since the program space is vast, as we evaluate empirically, it is safe to use a few trials to compute the LINT score without allowing the LLM to reconstruct the correct program by chance.

3 Caveats of LINT Score

When assessing the interpretability of programs, we assume a level of knowledge of the person interpreting them. LINT assumes the knowledge of an LLM, which may not reflect reality due to a mismatch of knowledge between the LLM and the target audience of the program. For example, if the goal of having interpretable programs is to teach people strategies for playing a real-time strategy game, then the LLM might have deeper knowledge of this genre of game than rookie players trying to learn strategies from the programs. As a result, a policy that is "interpretable" for the LLM is not necessarily

interpretable to the target audience. Conversely, if the program requires knowledge that the LLM does not possess (e.g., π is written in a DSL different from the languages with which the LLM is trained), LINT can produce false negatives.

Similarly to the BLEU score [Papineni *et al.*, 2002], LINT should not be used as an objective function. Using LINT as such could cause the system to disregard C, and the explainer could generate non-interpretable explanations. Instead of using it as a target, LINT can be used as a tool to assess the interpretability of computer-written programs, to bias design decisions made during the development cycle of synthesizers.

4 Empirical Methodology

The primary objective of our evaluation is to check whether the LINT scores correlate with the interpretability of a given set of programs. We rely on methods from the static obfuscation literature [Collberg and Nagra, 2009] to generate programs with different levels of interpretability. Static obfuscation algorithms have the goal of transforming a program before it starts running into less interpretable programs, with the goal of making it harder for adversarial agents to gain knowledge of the program by reading its implementation. For that, we consider semantics-preserving obfuscation transformations, where we can control the degree to which a program is obfuscated. We hypothesize that LINT scores correlate with the degree of obfuscation of a set of programs.

We consider two instances of LINT: one for evaluating the interpretability of programs that encode solutions to programming tasks; and another for evaluating programmatic policies [Mariño *et al.*, 2019] for playing MicroRTS, a real-time strategy game [Ontañón *et al.*, 2018]. **We provide the complete set of prompts used in our experiments in the Supplementary Materials.** All experiments used GPT-4 [OpenAI *et al.*, 2023]. We use $k = 5$ in all our experiments.

```
1 void subsets(char *av[], int c, int n,
2 char *sbset[], int sz) {
3     if (c == n) {
4         if (sz < n) {
5             for (int i = 0; i < sz; i++)
6                 printf(sbset[i]);
7             printf("----------");
8         }
9         return;
10    }
11    sbsets(av, c+1, n, sbset, sz);
12    sbset[sz] = av[c];
13    sbsets(av, c+1, n, sbset, sz+1);}
14
15 main(Q,O)char**O;{if(--Q){main(Q,O);O[Q]
16 [0]^=0X80;for(O[0][0]=0;O[++O[0][0]]!=0;)
17 if(O[O[0][0]][0]>0)puts(O[O[0][0]]);
18 puts("----------");main(Q,O);}}
```

Figure 2: Non-obfuscated code for computing proper subsets (lines 1–13); an obfuscated program for the same problem (line 15).

4.1 Classical Programming Problems

We consider 10 programs written in C for solving the following problems: computation of factorials, addition of two numbers, conversion of byte to binary, computation of all proper subsets of a set of arguments, of the value of π, of $\ln(n)$ for any n, of the smallest 100 prime numbers, of the square root of a number, sorting elements, and a program to play tic-tac-toe. The obfuscated versions of these programs were designed so that they would be as non-interpretable as possible, since all obfuscated programs we use are winning entries of the International Obfuscated C Code Contest [IOCCC, 1984]. The obfuscated programs were constructed using different techniques, such as replacing sequences of instructions with equivalents that are less interpretable [Cohen, 1993]. Figure 2 shows an example of the programs used in our experiment, where the first function is a non-obfuscated implementation for computing the proper subsets of a set of numbers, while the second is an obfuscated implementation to solve the same problem. The proper subsets of a set I include all subsets except I. The complete set of programs is provided in the Supplementary Materials.

The function B we consider in this experiment measures the number of input values that the reconstructed program correctly maps to their corresponding output value. A B-value of 1.0 indicates that the reconstructed program mapped all inputs to the correct output; a value of 0.0 indicates that the reconstructed program failed on all inputs.

4.2 Programmatic Policies

We also used programmatic policies for MicroRTS. These programs are categorized into two types: "synthesized" policies, written by a computer program in the domain-specific language known as the Microlanguage [Mariño *et al.*, 2019], and "human-crafted" policies, written in Java by human programmers. Both types of programs receive a state of the game and return the action the agent performs in that state.

We consider the two-player version of MicroRTS, where each player controls a number of units to collect resources, build structures, and train other units that will eventually battle the opponent. Programmatic policies are the current state of the art in this domain, with programmatic policies winning the last three competitions.[2] MicroRTS has the following types of unit: Worker, Light, Ranged, Heavy, Base, and Barracks. The first four types can move around a gridded map where the game is played and attack opponent units; Workers can collect resources and build Bases and Barracks; Bases can train more Worker units and store resources, while Barracks can train non-Worker units. Units differ in how much damage they can inflict on opponent units and in how much damage they can suffer before being removed from the game.

Microlanguage

The Microlanguage allows programs to iterate through all units the player controls, so it assigns an action to each of the units. The language also supports if-then-else structures. Figure 3 shows an example of a program synthesized with Local Learner (2L), a self-play algorithm [Moraes *et al.*, 2023]. The loops allow for an action prioritization scheme. This is because once an action is assigned to a unit, it cannot be overwritten by another action, so the instructions in the earlier for-loops will be assigned first. In the program shown in Figure 3,

[2] https://sites.google.com/site/micrortsaicompetition

```
1 for(Unit u)
2     for(Unit u)
3         u.train(Worker,Up,2)
4     u.attack_if_in_range()
5     u.train(Heavy,EnemyDir,8)
6 for(Unit u)
7     u.train(Light,Left,100)
8     u.build(Barracks,EnemyDir,1)
9     u.harvest(25)
10    u.attack(Closest)
```

Figure 3: Policy written in the Microlanguage.

training Worker units has the highest priority because the instruction for training these units is in the first nested for-loop to be executed (lines 2 and 3), which iterates through all units until it eventually finds a Base that will train them. Other actions that require the use of resources (e.g., constructing a Barracks in line 10), will be executed only if the player has enough resources after training Worker units.

Java

The human-crafted policies are written in Java and follow standard Java principles. This allows for the representation of more complex policies, but lacks the Microlanguage's domain-specific approach. Figure 4 shows an example.

```
1 for (Unit u : pgs.getUnits())
2     if (u.getType() == barracks
3     && u.getPlayer()== player
4     && gs.getActionAssignment(u) == null)
5         if (p.getResources() >= light.cost)
6             train(u, light);
```

Figure 4: Policy written in Java by human programmer.

Obfuscating Programmatic Policies

In the experiment with programmatic policies, we modified the programs to create different levels of interpretability, to verify whether the LINT scores correlate with these levels. We achieve this using the obfuscation technique of adding useless snippets to the programs, which is a known program obfuscation technique [Cohen, 1993]. We consider two levels of obfuscation: level 1, where we add a few lines of code that do not change the behavior of policy, and level 2, where we add a greater number of such lines compared to level 1. For the synthesized set, we add 10 and 23 lines for levels 1 and 2, respectively; for the human-crafted set, we add 38 and 71 lines for levels 1 and 2, respectively. Under the assumption that programs with longer useless snippets are less interpretable than programs with shorter snippets, we hypothesize that LINT assigns higher scores to non-obfuscated programs, lower scores to level 1, and the lowest scores to level 2.

Figure 5 shows a sample of a snippet that we add to the programmatic policies used in our experiments for level 1 of obfuscation; all snippets are shown in the Supplementary Materials, including level 2 snippets. The snippet in Figure 5 does not change the behavior of the policy because the only unit that can harvest resources is a builder, so line 6 does not change the behavior of the policy.

```
1 if (u.canHarvest()):
2     for (unit u)
3         if (u.isBuilder()):
4             pass
5         else:
6             u.harvest(50)
```

Figure 5: Sample of useless code snippet used in level 1.

Set of Policies Evaluated

For the synthesized set Π, we selected a subset of size 20 programs from the totally ordered set with approximately $1,000$ programmatic policies 2L synthesized for the BaseWorkers-16×16A map. Two adjacent policies in the ordered set are likely to be similar to each other due to the process in which 2L synthesizes them. We select 20 uniformly spaced policies from the ordered set to obtain a more diverse subset. That is, given that we have m policies in the ordered set, we select the policies with indices $\lfloor \frac{i \times (m-1)}{(19)} \rfloor$ with $i = 0, \cdots, 19$. For the human-crafted set, we used 10 programs selected from a collection available on GitHub.[3] We present all the programs used in our study in the Supplementary Materials.

Behavior Metrics

We used three behavior metrics B for programmatic policies. For all metrics, we consider a set of 10 policies, which are chosen from a totally ordered set of programmatic policies 2L synthesized; we refer to this set as the set of opponents $\mathcal{O}$. Although the policies evaluated and the set of opponents are selected from the same pool of programs, there is no overlap between the two sets. We ensure that our metric results are not skewed by having overly weak or overly strong opponents. This is achieved by, while sampling policies for $\mathcal{O}$, rejecting those that win or lose all matches against the set of 20 policies we evaluate in our experiment. Let $S_{\pi,o}$ be the set of states in which the policy π is to act in a match played with the opponent o in $\mathcal{O}$. Also, let $S_\pi = \bigcup_{o \in \mathcal{O}} S_{\pi,o}$ be the union of the states of all matches played with the opponents.

The first metric, which can be applied to any sequential decision-making problem with discrete action spaces, is the fraction in which the actions chosen by the reconstructed program π' match the actions chosen by the original program π for states in the set S_π: $|S_\pi|^{-1} \times \sum_{s \in S_\pi} \mathbf{1}[\pi(s) = \pi'(s)]$, where $\mathbf{1}[\cdot]$ is the indicator function. We refer to this metric as the **action metric**. If the reconstructed program is equivalent to the original, then the action metric is 1.0.

The second metric, which can be applied to any zero-sum game, compares the signature of wins, draws, and losses of the reconstructed policy with the signature of the original policy. The signature a_π of a policy π is a vector of size $|\mathcal{O}|$ where each entry i assumes the values of 1, 0, or -1, representing the result of a win, draw, or loss, respectively, of a match played between π and the i-th opponent in $\mathcal{O}$. This metric computes $|\mathcal{O}|^{-1} \times \sum_{i=1}^{|\mathcal{O}|} \mathbf{1}[a_\pi[i] = a_{\pi'}[i]]$, where $a_\pi[i]$

[3] https://github.com/rubensolv/SCV/tree/master/pvai

represents the i-th entry of a_π. We call this metric the **outcome metric**. Similarly to the action metric, if π' is equivalent to π, then the outcome metric value is 1.0.

The third metric compares the set of features observed in matches between the reconstructed program and $\mathcal{O}$ with the features observed in matches between the original program and $\mathcal{O}$. Let $F(\pi, o)$ be a vector of features observed in a match between π and o. We use the seven features of Aleixo and Lelis [2023], where each feature is the sum of the number of units of a given type that the player trained (or built) in all states of the match; the types can be Worker, Light, Heavy, Ranged, Base, or Barracks. A last feature sums up the amount of resources collected in the match. This metric measures the average normalized L1 norm between the feature vector of the original and reconstructed programs. We refer to this metric as **feature metric**. If the reconstructed program is equivalent to the original, then this metric is 0.0. While other metrics measure similarity, the feature metric measures dissimilarity.

We use these three metrics because each of them individually has weaknesses; together, they offer a more reliable summary of the behavior of a policy. Many of the actions in a MicroRTS match are related to Worker units collecting resources, so while two policies might encode totally different strategies, due to the large number of Worker units collecting resources, the policies could have a large action metric value. The outcome metric can also be misleading if almost any policy defeats the set of opponents or is defeated by the set of opponents (i.e., the opponents are too weak and/or too strong). Finally, the feature metric simply counts the number of units and resources, without measuring their behavior.

Baselines for Reconstructed Programs

We consider a number of programs as baselines for the programs LINT reconstructs. Namely, for each program π in Π, we compare the behavior metric values for the reconstructed program π' of π with a randomly selected program from Π that is different from π; we call this baseline **Rand**. In the experiment with the synthesized set, since all programs in Π were generated in a single run of 2L and for a fixed map, the programs Rand selects can be similar to π.

In another baseline, where we select a random program from the pool of programs 2L synthesizes for a different map; we use programs synthesized for the BaseWorkers-8×8 map with this baseline. Since the strategy for playing the game can change drastically from map to map, this randomly selected program is likely to be less similar to π than the programs Rand selects. We call this baseline **Rand-Other**.

Another baseline we consider selects the policy from the set of evaluated policies Π that is different from π but is most similar to π with respect to its syntax. We treat each line of a program as an element of a set. The program most similar to π is the one whose intersection with the syntax set of π is the largest. We refer to this baseline as **Closest-Syntax**.

The last baseline we consider returns the policy in Π different from the evaluated program π with the smallest L2 norm for the difference between its feature vector and the feature vector of π. We refer to this baseline as **Closest-Feature**.

Baseline for the Reconstruction System

The Reconstructor relies on the natural language explanation of the original program that the Explainer provides. To ensure that the program cannot be reconstructed by sampling programs without the explanation, we consider a baseline where we ask an LLM to generate a program for playing on a given map of MicroRTS. We provide a natural language description of the BaseWorkers-16×16A map and ask the model to generate a program written in the Microlanguage that can "play well" the game. We repeat this process k times, exactly how it is implemented in the reconstruction system for LINT, and we return the program with the best behavior score out of the k evaluated. We refer to this baseline as the **k-Shot LLM**.

5 Empirical Results: Programming Problems

The system was able to reconstruct all non-obfuscated programs to solve the programming problems and achieved the score of 1.0. The system was unable to reconstruct any of the obfuscated programs, resulting in a score of 0.0. As a representative example, Explanation 1 shows the Explainer's output for the non-obfuscated program shown in Figure 2.

Explanation 1 (Proper Subset - Non-Obfuscated)**.** *Imagine you have a list of items, and you want to figure out all the different ways you can select some of these items without taking all of them at once. When the program starts, it first checks if you've given it any items to work with. If not, it asks for some. Once it has the list of items, it begins to explore all the possible selections. It does this by starting with the first item and deciding whether to include it in the current selection or not. If it decides not to include the first item, it moves to the next item and makes the same decision. It continues doing this until it has considered every item in the list. After considering all the items, if it has left out at least one item from the current selection, it displays that selection for you to see. Then, it goes back and tries a different combination by including some items that it previously left out and leaving out some that it previously included. It continues this process until it has explored and shown you all possible selections of items from the list, always making sure it never shows you a selection that includes all the items at once. In the end, you're left with a display of all the different ways you can select some of the items without taking all of them.*

The explanation of the program is high-level and makes an analogy with a list of items, and the process of computing the proper subsets is described as a process of "selecting" and "displaying" the items, where all possible combinations of subsets are selected: *"It does this by starting with the first item and deciding whether to include it in the current selection or not"*. The Reconstructor can produce a correct implementation of a program that computes the proper subsets from this explanation. Explanation 2 shows the explanation LINT generated for the obfuscated program in Figure 2.

Explanation 2 (Proper Subset - Obfuscated)**.** *Imagine you have a list of items. Initially, all items are marked as "normal." The program goes through the following steps:*

1. *It starts at the end of the list and works its way to the beginning. For each item, the program does two actions:First, it keeps the item as "normal" and then displays all the items marked as "greater than normal." After displaying these items, it separates them with a line*

	Metrics B		
	Action (↑)	Outcome (↑)	Feature (↓)
LINT	0.940 ± 0.010	0.840 ± 0.042	0.133 ± 0.020
Rand	0.733 ± 0.015	0.615 ± 0.057	0.418 ± 0.018
Rand-Other	0.564 ± 0.025	0.470 ± 0.058	0.486 ± 0.012
Closest-Syntax	0.799 ± 0.015	0.600 ± 0.056	0.403 ± 0.018
Closest-Feature	0.823 ± 0.013	0.770 ± 0.049	0.189 ± 0.014
k-Shot LLM	0.343 ± 0.010	0.420 ± 0.057	0.441 ± 0.009

Table 1: Value of the behavior metrics for programmatic policies. Action and Outcome metrics are similarity metrics, so higher values are better (↑), while Feature is a metric of dissimilarity, so lower values are better (↓). The cells show the metric values and the 95% confidence interval.

Metric	Original Program	Obfuscation Level	
		Level 1	Level 2
Action	0.945 ± 0.010	0.866 ± 0.014	0.732 ± 0.012
Outcome	0.840 ± 0.042	0.705 ± 0.053	0.490 ± 0.058
Feature	0.133 ± 0.020	0.272 ± 0.024	0.418 ± 0.018

Table 2: Average values of the behavior metrics for the original program and for the two levels of obfuscation; the cells also show the 95% confidence interval.

of dashes. Second, it switches the item from "normal" to "greater than normal" and repeats the display process.

2. *After dealing with an item, the program moves to the next item closer to the beginning of the list and repeats step 2.*

This process continues until the program has considered all items in the list.

This explanation is well-structured and includes the steps that supposedly need to be performed. However, the description is not clear in some parts. For example, it is not clear what "greater than normal" means.

6 Empirical Results: Programmatic Policies

Table 1 shows the average and 95% confidence interval values for the 20 programs used in our experiment with the synthesized set. The LINT row shows the metric values computed for the original programs and their reconstructions. The baseline values represent the measurements between the original programs and the baseline programs. The values of LINT are the best according to all metrics, which shows that the reconstructed programs are more similar to the original than any of the baselines. Table 1 also shows that the baselines that obtain values closer to LINT are Rand, Closest-Syntax, and Closest-Feature. This is because these baselines select a program from the pool of programs 2L synthesized for the same map, and these programs tend to be similar to each other. Rand-Other obtained lower Action and Outcome values and a higher Feature value, demonstrating that the metrics can capture the expected differences between policies synthesized for playing in different maps. Finally, k-Shot LLM presents the lowest similarity values, demonstrating the importance of the explanation of the original program the Explainer generates.

Metric	Original Program	Obfuscation Level	
		Level 1	Level 2
Action	0.98 ± 0.003	0.92 ± 0.017	0.85 ± 0.031
Outcome	0.95 ± 0.036	0.84 ± 0.060	0.80 ± 0.066
Feature	0.07 ± 0.022	0.19 ± 0.029	0.22 ± 0.031

Table 3: Average values of the behavior metrics for the original program and for the two levels of obfuscation; the cells also show the 95% confidence interval.

Table 2 presents the results that test our hypothesis that LINT correlates with the degree of interpretability of the programs. The results indicate a higher similarity between the original and reconstructed programs for the policies 2L synthesizes than between the original obfuscated programs and their reconstructions. Also, LINT provides higher similarity scores and a lower dissimilarity score for Level 1 than for Level 2 obfuscation. Table 3 presents the results for the human-crafted set. These results align with those of the synthesized set, where the LINT-score decreases as we increase the level of obfuscation. Under the assumption that higher degrees of obfuscation result in less interpretable programs, the results support our hypothesis.

6.1 Representative Sample

In this section, we present the natural language explanation the Explainer generated for the program in Figure 3 (Explanation 3). Then, we present the program the Reconstructor generates from the explanation (Figure 6). This is a representative example of the programs used in the experiments.

Explanation 3 (Programmatic Policy)**.** *The script outlines a two-tiered strategy in the game of MicroRTS. It focuses on the tactical management of different types of units, ranging from Workers, to Heavy units, and finally Light units. The first part of the strategy involves a layered assignment of tasks. Every unit is asked to train two Worker units if possible. These Workers can be seen as the backbone of your army since they are the only units that can build structures, which in turn produce other units. At the same time, if a unit can't train a Worker, it is assigned to remain idle, thus serving as a defensive guard to your base. In addition, every unit is asked to train up to 8 Heavy units, that typically are stronger and can inflict more damage, acting as the frontline offensive unit. This phase essentially sets up your base with a balance of productive Workers and Heavy units.*

In the second part of the strategy, each unit is tasked with producing as many Light units as possible, up to a whopping 100. Light units are generally quicker and can swiftly react to the changes in the battlefield. Additionally, every unit is tasked to build a Barracks near the enemy's direction, effectively establishing a forward base closer to the enemy territory. They are also asked to harvest resources, up to 25 units, ensuring a steady supply for creating more units and structures. Finally, units are directed to attack the closest enemy units, marking an aggressive stance towards the opponent.

The overall goal of this strategy is to ensure a robust base with a mixture of Worker and Heavy units, while also maintaining an aggressive stance with a large army of Light units

```
1 for(Unit u):
2     u.train(Worker, Up, 2)
3     u.attack_if_in_range()
4     for(Unit u):
5         u.train(Heavy, EnemyDir, 8)
6     for(Unit u):
7         u.train(Light, Left, 100)
8     u.build(Barracks, EnemyDir, 1)
9     u.harvest(25)
10    u.attack(Closest)
```

Figure 6: Reconstruction of the program shown in Figure 3.

and Barracks near the enemy territory. It aims at resource gathering for continued production of units and structures, and pushing the opponent back through relentless attack.

Similarly to Explanation 1, the explanation describes the policy in a level of detail that allows for the reconstruction of a program that behaves identically to the original program according to our metrics. Figure 6 shows the reconstructed program. The reconstructed program is not identical in terms of line-by-line syntax, since information regarding the syntax is lost in the process of translating the program into a natural language explanation and back to a program. A casual inspection might even suggest that the reconstructed program does not behave as the original. This is because the original program has an instruction for training Worker units inside a nested loop, thus giving it the highest priority. The reconstructed program has training Worker units instruction inside the main loop. This means that the program can use the player's resources to assign actions to other units (e.g., train Light units in line 7) and by the time u is a Base in the outer loop, the player no longer has resources for training Worker units. However, a more careful inspection of the program reveals that, in the first states of the game, where the player trains Worker units, none of the actions that use resources can be assigned to a unit: the player cannot train Heavy and Light units (lines 5 and 7, respectively) because the player does not have a Barracks yet; the player cannot build a Barracks (line 8) because it does not have enough resources to do so. Thus, similarly to the original program, the reconstructed one prioritizes the training of Worker units.

7 Related Works

In contrast to the literature on programmatic policies, it is common to find evaluations of the interpretability of models in the context of supervised learning [Ribeiro *et al.*, 2016; Lundberg and Lee, 2017; Fong and Vedaldi, 2017]. We conjecture that the methodological difference between the programmatic policies and the supervised learning literature is due to the need of enlisting participants who understand both the application domain and computer programming for evaluating programmatic policies, while the latter only requires participants who understand the application domain.

Previous work in programmatic policies, such as NDPS [Verma *et al.*, 2018], Viper [Bastani *et al.*, 2018], Propel [Verma *et al.*, 2019], and π-PRL [Qiu and Zhu, 2022], describe systems that synthesize programmatic policies in the space of oblique decision trees [Murthy *et al.*, 1994]. Such trees represent programs with if-then-else structures with linear transformations of the inputs. Oblique decision trees are often assumed to be interpretable, which is likely true for small trees and low-dimensional problems, but it is unlikely to hold true for deep trees and high-dimensional problems.

Metrics to measure code understandability [Buse and Weimer, 2010; Posnett *et al.*, 2011; Daka *et al.*, 2015; Scalabrino *et al.*, 2016; Oliveira *et al.*, 2020] from the Software Engineering literature attempt to solve a related but different problem from the interpretability of programmatic policies we tackle in this paper. Code understandability considers scenarios where people write computer code that is meant to be understandable by other people, but it may not be because the person reading the code is not familiar with the API being used, the API documentation is lacking [Scalabrino *et al.*, 2021], or because the code is too long and time-consuming for one to understand. This is in contrast to our experiments on the interpretability of policies, where we assume that one has access to the correct resources (e.g., API description) and enough time to interpret a policy. In the interpretability of policies, we are interested in evaluating programs generated by other programs with the goal of maximizing the agent's reward and not necessarily to be interpretable.

Recent work showed that there is currently no effective metric to measure code understandability [Scalabrino *et al.*, 2021]. This contrasts with our results, which suggest that LINT can be used as a reliable metric to assess the interpretability of policies. Although the two problems have nuanced but important differences, our encouraging results on policy interpretability suggest that future research could investigate the use of LLMs to measure code understandability.

8 Conclusions

Programmatic policies are often synthesized with the expectation of interpretability. However, to our knowledge, there has not been a systematic evaluation of the interpretability of such policies, probably due to the cost associated with such an evaluation. Especially because the evaluation of programmatic policies might require human users proficient in computer programming. In this paper, we presented an inexpensive methodology based on LLMs to assess the interpretability of programmatic policies. Namely, we introduced the LLM-based Interpretability (LINT) score for programs. The LINT score of a program is computed by having an LLM generate a description of it in natural language, which is provided as input to another LLM. This second LLM tries to reconstruct the program from the natural language description. The LINT score measures the similarity between the original and reconstructed programs. Our empirical evaluation of LINT relied on the literature on program obfuscation and we assumed that obfuscated programs are less interpretable than non-obfuscated ones. Empirical results on programming problems and programmatic policies showed that the LINT scores of the evaluated programs correlate with their interpretability. Our results suggest that LINT can be used as a tool to assess the interpretability of programmatic policies.

References

[Aleixo and Lelis, 2023] David S. Aleixo and Levi H. S. Lelis. Show me the way! Bilevel search for synthesizing programmatic strategies. In *Proceedings of the AAAI Conference on Artificial Intelligence*. AAAI Press, 2023.

[Bastani *et al.*, 2018] Osbert Bastani, Yewen Pu, and Armando Solar-Lezama. Verifiable reinforcement learning via policy extraction. In *Proceedings of the International Conference on Neural Information Processing Systems*, pages 2499–2509. Curran Associates Inc., 2018.

[Brown *et al.*, 2020] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[Buse and Weimer, 2010] Raymond P. L. Buse and Westley R. Weimer. Learning a metric for code readability. *IEEE Transactions on Software Engineering*, 36:546–558, 2010.

[Cohen, 1993] Frederick B. Cohen. Operating system protection through program evolution. *Computer Security*, 12(6):565–584, 1993.

[Collberg and Nagra, 2009] Christian Collberg and Jasvir Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley Professional, 2009.

[Daka *et al.*, 2015] Ermira Daka, José Campos, Gordon Fraser, Jonathan Dorn, and Westley Weimer. Modeling readability to improve unit tests. In *Proceedings of the Joint Meeting on Foundations of Software Engineering*, page 107–118. Association for Computing Machinery, 2015.

[Fong and Vedaldi, 2017] Ruth C Fong and Andrea Vedaldi. Interpretable explanations of black boxes by meaningful perturbation. In *Proceedings of the IEEE International Conference on Computer Vision*, 2017.

[Inala *et al.*, 2020] Jeevana Priya Inala, Osbert Bastani, Zenna Tavares, and Armando Solar-Lezama. Synthesizing programmatic policies that inductively generalize. In *International Conference on Learning Representations*, 2020.

[IOCCC, 1984] IOCCC. International obfuscated c code contest, 1984. Accessed: 2023-08-11.

[Lundberg and Lee, 2017] Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In *Advances in Neural Information Processing Systems*, 2017.

[Mariño *et al.*, 2019] Julian R. H. Mariño, Rubens O. Moraes, Claudio F. M. Toledo, and Levi H. S. Lelis. Evolving action abstractionsfor real-time planning in extensive-form games. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2019.

[Medeiros *et al.*, 2022] Leandro C. Medeiros, David S. Aleixo, and Levi H. S. Lelis. What can we learn even from the weakest? Learning sketches for programmatic strategies. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 7761–7769. AAAI Press, 2022.

[Moraes *et al.*, 2023] Rubens O. Moraes, David S. Aleixo, Lucas N. Ferreira, and Levi H. S. Lelis. Choosing well your opponents: How to guide the synthesis of programmatic strategies, 2023.

[Murthy *et al.*, 1994] Sreerama K. Murthy, Simon Kasif, and Steven Salzberg. A system for induction of oblique decision trees. *Journal of Artificial Intelligence Research*, 2(1):1–32, 1994.

[Oliveira *et al.*, 2020] Delano Oliveira, Reydne Bruno, Fernanda Madeiral, and Fernando Castor. Evaluating code readability and legibility: An examination of human-centric studies. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 348–359, 2020.

[Ontañón *et al.*, 2018] Santiago Ontañón, Nicolas A. Barriga, Cleyton R. Silva, Rubens O. Moraes, and Levi H. S. Lelis. The first microrts artificial intelligence competition. *AI Magazine*, 39(1), 2018.

[OpenAI *et al.*, 2023] OpenAI, :, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mo Bavarian, Jeff Belgum, Irwan Bello, and ... Barret Zoph. Gpt-4 technical report, 2023.

[Papineni *et al.*, 2002] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318. Association for Computational Linguistics, 2002.

[Posnett *et al.*, 2011] Daryl Posnett, Abram Hindle, and Premkumar Devanbu. A simpler model of software readability. In *Proceedings of the Working Conference on Mining Software Repositories*, page 73–82. Association for Computing Machinery, 2011.

[Qiu and Zhu, 2022] Wenjie Qiu and He Zhu. Programmatic reinforcement learning without oracles. In *International Conference on Learning Representations*, 2022.

[Ribeiro *et al.*, 2016] Marco T Ribeiro, Sameer Singh, and Carlos Guestrin. "why should i trust you?" explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2016.

[Scalabrino *et al.*, 2016] Simone Scalabrino, Mario Linares-Vásquez, Denys Poshyvanyk, and Rocco Oliveto. Improving code readability models with textual features. In *IEEE International Conference on Program Comprehension*, pages 1–10, 2016.

[Scalabrino *et al.*, 2021] Simone Scalabrino, Gabriele Bavota, Christopher Vendome, Mario Linares-Vásquez, Denys Poshyvanyk, and Rocco Oliveto. Automatically

assessing code understandability. *IEEE Transactions on Software Engineering*, 47(3):595–613, 2021.

[Verma *et al.*, 2018] Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. Programmatically interpretable reinforcement learning. In *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 5045–5054. PMLR, 2018.

[Verma *et al.*, 2019] Abhinav Verma, Hoang M. Le, Yisong Yue, and Swarat Chaudhuri. Imitation-projected programmatic reinforcement learning. In *Proceedings of the International Conference on Neural Information Processing Systems*. Curran Associates Inc., 2019.