FALAA: Framework for the Abstraction of Language Agent Architectures

Nicolas Brandstetter[1,2,3,4], Felipe Bravo-Marquez[1,3,4][0000−0002−2153−4306], and Federico Olmedo[1,3][0000−0003−0217−6483]

[1] Department of Computer Science (DCC), University of Chile, Chile
[2] Department of Electrical Engineering (DIE), University of Chile, Chile
[3] Millennium Institute for Foundational Research on Data (IMFD), Chile
[4] National Center for Artificial Intelligence (CENIA), Chile
{nicolasbrandstetter@ug,fbravo@dcc,folmedo@dcc}.uchile.cl

Abstract. The rapid growth of the language agent field, driven by advances in Large Language Models (LLMs), has led to agent designs that often rely on *ad hoc* methods. This lack of structure makes it challenging to understand, compare, reuse, and evolve these agents effectively, highlighting the need for a standardized framework to describe their architectures. This paper introduces `FALAA` (*Framework for the Abstraction of Language Agent Architectures*), a dual-level specification framework aimed at addressing these challenges by proposing a standardized structure and a description methodology that abstracts and describe LLM-based agent architectures using six essential components: `Planner`, `Executor`, `Evaluator`, `Reflector`, `Memory`, and `Environment`. Using this proposed structure, `FALAA` leverages UML (Unified Modeling Language) and OCL (Object Constraint Language) to provide a description methodology composed by two levels: a (1) *conceptual description level*, which visually represents the standardized components and behaviors of language agents through UML class and sequence diagrams, and a (2) *formal specification level*, which employs OCL to define invariants, conditions, and complex behaviors beyond UML's expressive capacity. By establishing a clear convention for the structure and responsibilities of essential agent's components hence along with a standardized description methodology, `FALAA` aims to eliminate ambiguity and ensure a reusable, unified standard for agent architectures. This framework pursue the goal of improved clarity, consistency, and precision in describing language agents, thereby supporting better comparison, evaluation, and development of LLM-based agents. The proposed approach is exemplified through a practical example and case studies, demonstrating its effectiveness in representing agent behaviors and architectures.

Keywords: Language Agents · Large Language Models · Framework · UML · OCL · Agent Architecture · Formal Specification.

1 Introduction

Language agents constitute a category of autonomous intelligent agents built upon large language models (LLMs), and are defined by their capacity to per-

ceive, act, reason, learn, and memorize [1,3,8]. The development of advanced models such as GPT-4 [6] and LLaMA [12] has positioned these agents as effective solutions for complex tasks, offering a viable alternative to traditional task-specific fine-tuned models [2]. By exploiting the reasoning capabilities and general knowledge embedded in LLMs, they achieve robust performance in diverse tasks through mechanisms such as in-context learning and instruction tuning, which enable generalization without parameter adjustment. As a result, the field is rapidly evolving, with new architectures emerging continuously to harness the growing potential of these models [2].

The recent emergence of the LLM field, combined with the accelerated development of new language agent architectures, has resulted in their design and implementation being largely ad hoc, without adherence to any general or standardized methodology [2]. Consequently, each researcher typically defines an agent's behavior using a mixture of natural language, mathematical notation, pseudocode, explicit code, and diagrams [10,17,13].

However, the use of natural language to describe constraints and behaviors often introduces ambiguities, hindering a precise understanding of a language agent's internal mechanisms [5,18]. This lack of clarity becomes particularly critical when selecting or comparing agents for specific tasks, as each architecture exhibits unique strengths and limitations. In the absence of a standard framework, users are forced to interpret and re-implement agent behaviors from scratch, increasing development time and reducing component reusability [2].

Ultimately, the core issue lies in the absence of a standardized structure and shared terminology for describing language agent architectures. This gap restricts comprehension, complicates development and scalability, and hinders effective comparison and reuse of agent designs.

To address this problem, several works [1,11,2,14] have proposed frameworks for describing language model-based agents, often focusing on internal functionalities or formal specifications. However, these efforts still fall short in fully resolving ambiguities related to terminology, component organization, and structural representation—gaps that motivate the framework proposed in this work.

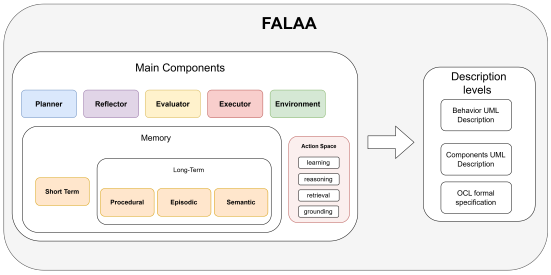

Fig. 1: Glimpse of the essential components and levels of the methodology proposed by `FALAA`.

As a result, we introduce `FALAA`: a Framework for the Abstraction of Language Agent Architectures. As illustrated in Figure 1, `FALAA` establishes a standardized structure that abstracts LLM-based agent architectures into six essential components: `Planner`, `Executor`, `Evaluator`, `Reflector`, `Memory`, and `Environment`. Following the approach of [11], the memory component is further subdivided into four types: `Short-term memory` and the long-term memories `Semantic`, `Episodic`, and `Procedural`.

Furthermore, to classify the actions performed by the internal components of a language agent, `FALAA` decomposes the action space into three categories of internal actions—*Learning*, *Reasoning*, and *Retrieval*—and one category of external actions, referred to as *Grounding* actions.

In order to provide a guide, `FALAA` gives a methodology to describe the architecture and behavior of a language agent, which is composed of two description levels: a visual-conceptual abstraction level where the components and behavior of the agent are described using UML class and sequence diagrams, and a formal specification level which employs OCL to define invariants, conditions, and complex behaviors beyond UML's expressive capacity, aimed to avoid, as far as possible, the dependency to a natural language description.

Our *Contributions* are summarized as follows: We present a new, general structure and terminology for describing language agents in a clear and unambiguous way, based on a set of components aligned with the capabilities these agents should possess. Furthermore, we propose a methodology for describing language agents using our proposed convention, aiming to avoid ambiguities and clearly outline their behavior and structure by leveraging two well-established tools: UML and OCL.

This paper is structured as follows. Section 2 presents frameworks that propose approaches that address current difficulties in characterizing language agents. Section 3 describes `FALAA`, explaining each essential component and present the proposed methodology for describing agents at a visual-conceptual and formal specification. In section 4, we use `Reflexion` [10] and `Retroformer` [17] as case studies because of their similarities and differences between their architectures. Each language agent is briefly described, followed by some ambiguities addressed by describing them using `FALAA`. Then, a brief comparison between these formalized architectures is presented. Finally, section 5 summarizes the performance of `FALAA` and presents possible improvements and future work.

2 Related Work

The idea of having a standard *framework* for the formalization and explanation of the behavior of language agents is not new. In the literature, various approaches have been proposed, ranging from the abstraction of agent's components based

on their internal functionalities to representing language agents as finite state machines.

2.1 Frameworks for Agent-based Language Models Based on Internal Functionalities

Given the proliferation of new language agent architectures, several studies have proposed frameworks that describe their composition based on internal functionalities. For instance, [1] identifies essential components such as *planning, memory, rethinking, execution, tool usage*, and different *interaction environments*. In a similar vein, [14] introduces a modular model with specific roles: the *profile module*, *memory module*, *planning module*, and *action module*.

Other approaches adopt cognitive architecture principles, incorporating elements inspired by human reasoning. The `CoALA` framework [11], for example, organizes agents around three axes: a memory system (with short-term and various long-term memories), an internal and external action space, and a decision-making process encompassing planning and execution. Likewise, [16] presents a framework based on three functional areas: the *Brain* (which includes the LLM and manages reasoning, memory, and actions), the *Perception* module (which translates sensory input into LLM-interpretable data), and the *Action* module (which performs decisions via text generation, tool usage, or physical interaction).

These frameworks contribute significantly to defining a standard structure for language agents by identifying core functionalities and modular divisions. However, despite these advances, previous frameworks do not fully address the ambiguities that arise when describing agent architectures. Their focus remains on functional decomposition rather than on providing an unambiguous, intuitive, and visual representation that facilitates the comparison and evaluation of different agents. In contrast, `FALAA` aims to fill this gap by offering not only a standardized structure and terminology, but a formal, visual, and standardized way to describe and analyze language agent architectures.

2.2 Linear Temporal Logic (LTL) for Agent-based Language Models Descriptions

To address the lack of standardization in the structure of language agents, [2] proposes a declarative framework that formally specifies high-level agent behavior. The authors point out that many architectures are designed in an *ad hoc* manner, which complicates performance comparison across tasks. They also note that describing agents directly through code often leads to rigid behavior, limiting the flexibility of LLMs in selecting optimal actions.

Their framework is structured around three modules: a *text generator*, a *decoding monitor*, and a *correction module*. The agent begins in an initial state s_0,

which is passed to the text generator. This module produces outputs either from the LLM or its environment, which are then validated by the decoding monitor. The monitor verifies whether the generated output satisfies a logical specification expressed in Linear Temporal Logic (LTL) and Lisp-style S-expressions. If the expected state s_{end} is reached, the process concludes; otherwise, the correction module modifies the output to align it with the specification and reintroduces it into the generation cycle.

While this proposal contributes to reducing ambiguity through formal specification, its generality limits its ability to represent key architectural features of certain agents. For instance, elements like long-term memory in agents such as *Reflexion*, which plays a central role in storing internal reflections, are not explicitly modeled when using this framework. In contrast, the approach proposed in this thesis seeks to offer a visual and structural formalization capable of capturing such architectural distinctions.

In summary, `FALAA` aims to address these problems by proposing not only a standard terminology and responsibilities for the components of a language agent but also offering a guide for describing these agent's components and behavior through the explicit use of UML Class and Sequence diagrams, accompanied by formal specifications using OCL to avoid ambiguities in critical areas.

3 FALAA: Framework for the abstraction of language agents architectures

In the current section, we first define the essential components of a language agent´s architecture proposed by `FALAA`, including the classification of the both internal and external action space of the agent, which will established a common structure for the description of LLM-based agents. Finally, we will characterize the methodology given by `FALAA` to describe the architecture and behavior of a language agent under the standard proposed, which is composed by two description levels: a conceptual and visual description level using UML Class and Sequence diagrams, and a formal specification level throw OCL.

3.1 Essential components of a language agent

`FALAA` defines a set of essential components inspired by the work of [1], which characterizes language agents as entities composed of *planning, memory, reflection, execution, tool usage*, and *interaction environments*. Building on this characterization, the proposed framework encapsulates the structure of LLM-based agents into six core components: `Planner`, `Executor`, `Evaluator`, `Environment`, `Reflector`, and `Memory`. Both the action space and the agent's memory are internally classified following the structure proposed in [11], which distinguishes four memory types—`short-term`, `episodic`, `semantic`, and `procedural`—as well as four action categories: *grounding, reasoning, retrieval*, and *learning*. While the

framework allows for the inclusion of additional components when necessary to model specific behaviors, these should not override the core responsibilities assigned to the essential components.

The selection of these components is grounded in both common patterns observed across existing language agent architectures and the broader definition of LLM-based agents as autonomous entities that interact with and act upon an environment, make decisions, learn, reason, and maintain memory [9,4,15].

Action space. It is defined as the set of actions a language agent can execute both externally in an environment and internally within the agent itself. Drawing inspiration from the work of [11], the actions in this space are classified as:

- **Grounding**: External actions to interact with the environment.
- **Reasoning**: Internal actions to reason and generate new information from short-term memory.
- **Retrieval**: Internal actions to retrieve relevant information from long-term memories into short-term memory.
- **Learning**: Internal actions to learn from past experiences, updating long-term memories.

Memory. Stores and provides access to information relevant to the agent. Referencing the proposal by [11], it is divided into:

- **Short-term memory**: Stores immediate-use information (current state, problem, proposed action). In implementations, this abstraction is sometimes represented as local variables in the code.
- **Episodic memory**: Records past experiences (actions, completed/failed tasks).
- **Semantic memory**: Contains knowledge and reflections for decision-making (summaries, learnings).
- **Procedural memory**: It abstracts the information used to execute actions or guide the agent's behavior. It stores two types of information: implicit and explicit. The implicit information stored in `procedural memory` refers to data such as that stored in the parameters of a large language model, which are not explicitly presented. Explicit information ranges from instructions for a given LLM, the external action space allowed by the environment, or even new skills acquired by the agent over time. A difference from the definition by [11] is that codes and functions that produce actions (described in the action space) are not considered part of `procedural memory` in this work; instead, these methods themselves are considered part of the agent's internal action space.

Both `semantic, episodic and procedural memory` must contain methods that allow for as many retrieval actions as necessary to provide their stored information.

Planner. Uses an LLM to reason about a problem and generate an action plan. It employs prompting techniques —such as *Chain of Thought* or `ReAct`— and reads instructions from long-term and short-term memories to contextualize its LLM, proposing *grounding* actions that are stored in the trajectory from `short-term memory`. It must contain at least a method that executes a *reasoning action* aim to generate the action that will be returned.

Executor. Executes the *grounding* actions suggested by the `Planner`, interacting with the environment and receiving observations from it, storing them in the trajectory in the `short-term memory`, all by its own *grounding action* method.

Environment. An external component that allows the execution of actions and the reception of observations or rewards. It can be as simple as a QA environment or as complex as the one designed by [13] based on MINEFLYER, focused on interaction with Minecraft.

Reflector. Performs *reasoning* actions after the execution of the action proposed by the `Planner`, generating feedback about what happened. Using its instantiated LLM, it produces reflections or critiques that can be stored as knowledge or used to improve future actions proposed by the `Planner`. Occasionally, the `Reflector` is also used as an evaluator of the executed action, providing, in addition to feedback, a signal indicating the quality of the action executed by the `Executor`.

Evaluator. Behavior related to evaluating actions and behaviors performed by the agent, or validating outputs generated by other components, can be encapsulated in an `Evaluator` component. This component is responsible for evaluating and providing a signal indicating the quality of the generated action or behavior by its *reasoning action* methods.

3.2 Description methodology

The description methodology given by `FALAA`, which proposes a structured way to describe and specify an LLM-based agent´s architecture, is composed by two complementary levels:

(1) Conceptual Description Level.

The main objective at this level is to offer a high-level understanding of the agent´s architecture without ambiguity or excessive detail, clarifying relationships and behaviors in a visual-structured way. The conceptual description level should provide a concise, visual overview of the system in the standard structure proposed.

At this level, we defined the principal components and their interactions using UML Class and Sequence diagrams. By examining both diagrams, designers can quickly grasp the agent's architecture, its primary interactions, and the way in

which its data and operations are logically grouped. On the other hand, following the standard structure proposed by `FALAA`, designers will be able to compare sections of their agents with other agents described in the literature, facilitating the understanding and comparison of behaviors and architectures of LLM-based agents.
Here is a general guideline for applying this level:

- *Identify core components:* Determine the essential building blocks of the agent (e.g. `Planner`, `Reflector`, `Executor`, `Evaluator`, various memories, and any external environment). Each essential component (proposed by `FALAA`) should mantain the original name, to ensure a convention is followed. Each component should be assigned a specific responsibility. Additional components can be defined if necessary to fulfill specific behaviors, but they should not replace the responsibilities of the essential components.

- *Class Diagram Abstraction:* Represent each component as a class, identifying attributes and methods that show the component's functionalities. Indicate relationships such as *composition*, *aggregation*, and *associations* to clarify how components depend on one another or store references to one another. This diagram also indicates the nature of each method (e.g., *reasoning*, *grounding*, *retrieval*, *learning*), making explicit what type of action or process is being performed. It is not necessary to assign an action type to each method, but it is encouraged to do so at least for the main functionalities.

- *Sequence Diagram Abstraction:* Model the behavior among components. Show how the agent initializes, receives tasks, proposes actions, interacts with the environment, evaluates potential results, updates memory structures and every behavior that is crucial for the agent's operation. Emphasize repeated processes (e.g., work cycles or loops) and decision points (e.g. whether a reflection it has to be generated). This helps to reveal how the system behaves step by step and how information flows among components. It is encouraged to use the various sequence interaction fragments given by UML, such as *loop*, *alt*, *opt*, *par* to represent the agent's behavior more clearly and precisely. It is also encouraged to use the *ref* fragment to avoid extensive and repetitive sequence diagrams.

(2) Formal Specification Level. While UML diagrams capture the main structure and dynamics, some behaviors require a precise and unambiguous definition. The formal specification level complements the conceptual one by providing a rigorous means of describing and verifying behaviors that diagrams alone cannot capture. Designers choose which behaviors to formalize based on the system's complexity and the potential for misunderstandings. Commonly, behaviors that are pivotal to the agent's operation—such as memory handling or task verification—are prime candidates for OCL specifications. For such scenarios, this methodology incorporates the use of `OCL` constraints to formalize

critical invariants, preconditions, and postconditions[5]. This level in `FALAA` is a complement and shouldn't be seen as a alternative to the conceptual description level.

Below is a guideline for this level:

- *Identify critical behaviors:* Determine which aspects of the agent's logic could lead to ambiguity, could´nt be described in UML diagrams or are crucial for correctness (e.g., ensuring a specific sequence of actions and observations, conditions that define when an answer is considered complete, or methods that reset internal states).
- *Define invariants:* State the conditions that must always remain true for the system (e.g., the order in which actions and observations must alternate within a trajectory).
- *Express preconditions and postconditions:* For each significant operation, specify the required inputs or states (preconditions) and the resulting modifications (postconditions). This clarifies the exact responsibilities of a method and prevents unintended side effects, something important because, not always the functionalities assigned to a component are obvious, and can be interpreted erroneously.
- *Maintain consistency with UML:* OCL integrates naturally with the UML model. Each constraint refers to elements from the Class Diagram (attributes, methods, relationships), so its necessary to ensure that the constraints are consistent with the elements shown in the UML diagrams.

In summary, by combining the conceptual description level (UML diagrams) with the formal specification level (OCL), this methodology provides a comprehensive guide to accurately describe and implement LLM-based agents. The designer ultimately decides how extensively to specify each behavior, ensuring that critical aspects are addressed without overcomplicating the model.

4 Analysis of Language Agents using FALAA

In this section, we analyzed two architectures of agents based on large language models: `Reflexion` and `Retroformer`. These agents were selected due to their relevance in the field of language agents, as well as the similarities and differences they exhibit. Both—`Reflexion`, and `Retroformer`—are designed to tackle reasoning and decision-making tasks, and have been evaluated in the same set of environments. Notably, `Retroformer` builds upon the `Reflexion` architecture, inheriting several components and behaviors. This makes them particularly well-suited for comparison under a unified descriptive framework.

[5]Each OCL constraint could be named. Naming invariants or pre/postconditions is not mandatory in OCL, but it is considered good practice to facilitate the understanding of the specification.

The objective is to demonstrate how the standard proposed by `FALAA` can be used to describe each of these architectures clearly and uniformly, identifying ambiguous points, gaps, or inconsistencies in the original documentation.

Throughout the following sections, the key aspects of each agent will be reviewed, both in terms of structure and behavior (execution cycle, action and reflection generation, reward evaluations, among others). It will be illustrated how descriptions in natural language often lead to different or incomplete interpretations, and how formalization under `FALAA` helps to clarify and unify these aspects.

Finally, a brief comparative analysis of the two agents will be provided, highlighting their similarities and differences both conceptually and operationally[6]. This approach aims to demonstrate the versatility of `FALAA` and its potential to facilitate the understanding and design of architectures for agents based on language models.

4.1 Reflexion

`Reflexion` [10] is an agent designed to enhance LLM-based systems through linguistic feedback after task failures. Upon failure, the trajectory of actions and observations generated during task solving is summarized into textual reflections that guide future attempts. Its structure defines three main components: an *Actor*, responsible for action generation; an *Evaluator*, which assesses trajectories by assigning rewards or ratings; and a *Self-reflection* module, which generates corrective feedback. Additionally, short-term memory (referred to as *trajectory*) and long-term memory (*Mem*) are defined to store recent experiences and accumulated reflections, respectively.

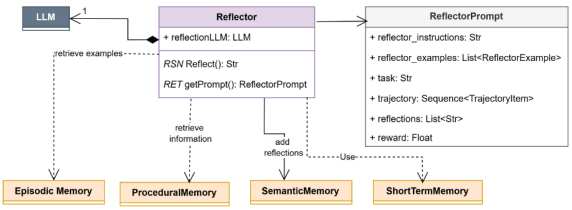

Fig. 2: UML class diagram of the reflector component of the Reflexion agent.

A key element of `Reflexion` is the *Self-reflection* module. Originally described mainly in natural language, this component relies on a dedicated LLM instance that, when the *actor* module failed in solve the current task, is prompted and generates new reflections. The prompt includes the current task, the agent's trajectory, previous reflections, instructions, examples, and a reward signal produced by the `Evaluator`. New reflections are stored in long-term memory, with a

[6]It is worth noting that, to avoid overloading the paper, the full specification of `Retroformer` architecture under `FALAA`—including detailed diagrams and formal constraints—is presented for those interested in https://dccuchile.github.io/FALAA/

constraint that limits the total number to three, addressing the context window limitations of LLMs.

Under `FALAA`, the *self-reflexion* module is formally modeled which is associated to the `Reflector` component, as shown in the UML class diagram in Figure 2. The class `Reflector` includes an LLM instance and is associated with the `ReflectorPrompt`, which encapsulates the structure of the input prompt listed above. The reflection process is captured by the *Reflect()* method, whose execution is detailed in the sequence diagram of Figure 3. The method's behavior is formally specified in OCL: the precondition 2 formally enforces that a new reflection can only be added if fewer than three are currently stored, while the invariant 1 ensures that the memory never exceeds this limit.

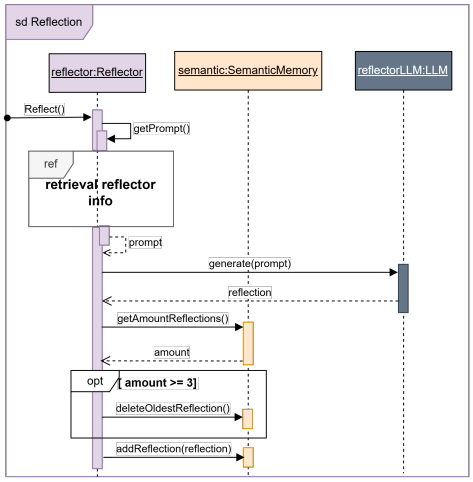

Fig. 3: UML sequence diagram of the reflection process of the Reflexion agent.

$$\textbf{context } \texttt{SemanticMemory } \textbf{inv } \text{MaxThreeReflections}: \quad \text{self.reflections} \rightarrow \text{size}() \leq 3 \tag{1}$$

$$\textbf{context } \texttt{SemanticMemory} :: \text{addReflection}(\text{newReflection} : \text{Reflection}): \text{OclVoid} \quad \textbf{pre}: \quad \text{self.reflections} \rightarrow \text{size}() < 3 \tag{2}$$

4.1.1. Ambiguities

`Reflexion` is described by the authors using both natural language, diagrams, and pseudocode which provide a high-level overview of their proposal. However, due to the lack of a standardized structure, ambiguities arise in its definition, which are addressed by describing `Reflexion` using `FALAA` as shown below with two examples.

1) Scope of the *Actor* Component. A key challenge lies in interpreting the concept of an *Actor*. According to [10], the *Actor* uses an LLM to produce text and actions, interacts with an environment, and forms a trajectory over multiple steps in a work cycle. Since it generates actions, reasons about them, and accesses a memory (though that memory belongs to the broader `Reflexion`

agent rather than the *Actor* itself), it is not fully accurate to label this component as a standalone "agent", because, under the language-agent structure established by `FALAA`, an agent must include its own memory.

By describing `Reflexion` within `FALAA`, the `Actor` component is defined as an additional composite component which encompasses both the `Planner` and `Executor` components, managing the entire process of attempting to solve a given task.

Using the above, when `Reflexion` mentions that other agents can be used as the Actor [10], it implies that certain elements—such as the prompting technique used by the `Planner` or the evaluation method implemented by the `Evaluator`—can be replaced with their respective counterparts from other agents.

This perspective helps clarify the confusion noted in [2], which treated `Reflexion` as an extension of `ReAct`—another agent arquitecture which Reflection´s authors uses as the actor in their experiments—.

2) Semantic Memory and Learning Limitations. A notable insight gained from applying `FALAA` is rely to `Reflexion` retains only the last three reflections in its long-term memory (see figure 3). Although [10] briefly mentions this limitation, formalizing the agent's behavior highlights how this storage constraint—described in the `FALAA`´s formal specification level, specification 1 and 2— prevents more extensive accumulation of knowledge. In other words, `Reflexion` cannot effectively learn from older tasks once it exceeds three stored reflections. This underscores a potential enhancement for future LLM-based agents aiming to incorporate deeper, more persistent learning.

4.2 Retroformer

`Retroformer` [17] extends the `Reflexion` architecture by incorporating reinforcement learning techniques to enhance the quality of its reflections. Its main contribution lies in the use of *RLHF* [7] to train an auxiliary neural network, responsible for evaluating the quality of the reflections generated by the agent's retrospective model. After training, this reward model is used to guide the reflection process: during each iteration, multiple candidate reflections are generated using a *best-of-n sampling* strategy, and the reward model selects the best reflection, aiming to maximize the agent's overall performance.

`Retroformer` was originally described by [17] through three main components: the *Actor*, the *Retrospective Model*, and the *Memory*, which includes a *replay buffer* for storing training data. Under the `FALAA` standard, these elements are mapped more precisely: the *Actor* integrates both the `Planner` and the `Executor`, as in the case of `Reflexion`; the *Retrospective Model* is associated with the `Reflector`, responsible for generating reflections; and the *Memory* is divided into `FALAA`´s short-term and long-term memories with the *replay buffer*

specifically categorized under `episodic memory`.

4.2.1. Ambiguities

Despite this more structured mapping, `Retroformer` exhibits greater architectural complexity than `Reflexion`, incorporating new components and processes. However, the absence of a standard descriptive framework led to an *ad hoc* presentation that mixes mathematical formalism with natural language explanations. As a result, ambiguities arise in key concepts—such as the reflection generation process—allowing for multiple interpretations of the original proposal.

1) Reflection Process in `Retroformer`.

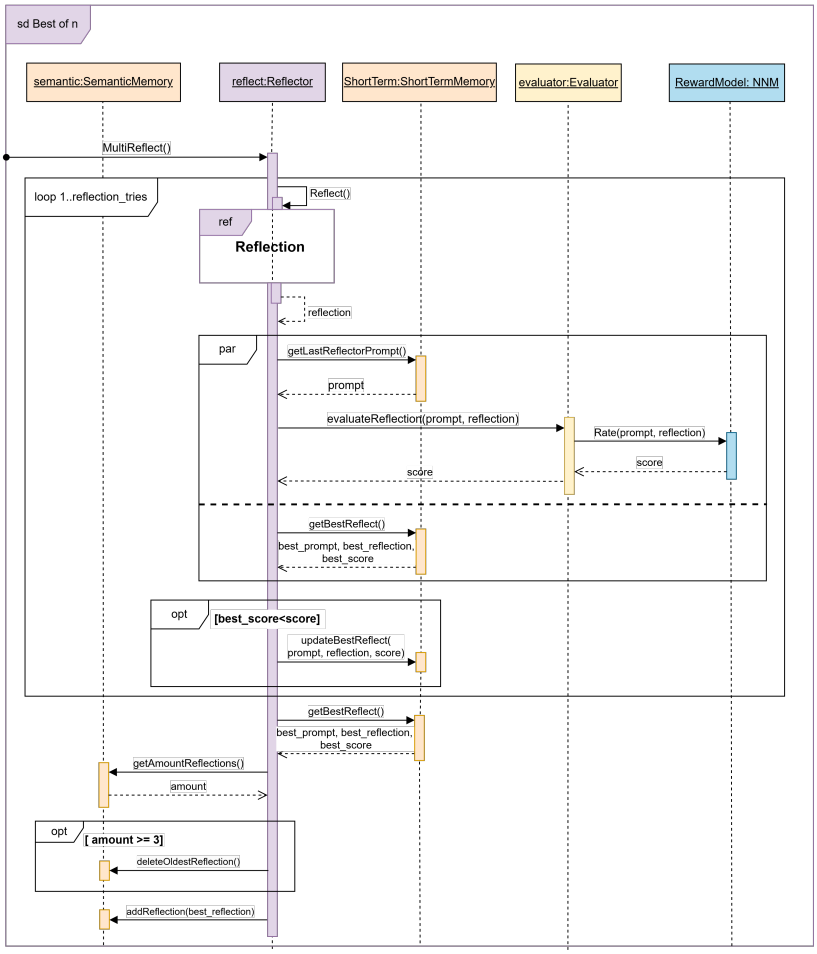

Fig. 4: UML sequence diagram of the best reflection selection process for the `Reflector` component of a `Retroformer` agent.

The essential innovative aspect of `Retroformer` lies in its reflection process, which is based on generating multiple reflections when the agent fails a task and then selecting the best one using a pre-trained reward model. Although [17] provides more detail in describing the training of the reward and retrospective models, it does not comprehensively specify how the reflection process unfolds during the agent's normal execution.

In the description of `Retroformer` using `FALAA`, this mechanism is explicitly defined (as seen in figure 4), reducing the likelihood of misinterpretations and facilitating an understanding of how reflections are generated and selected during a typical work cycle. In the `FALAA` description, it can be observed that, after detecting a failed task, the `Reflector` iterates a number of times defined by the developer, generating a reflection in each iteration (process similar to Reflexion's, presented in figure 3), which the `Evaluator` evaluates using the reward model.

The score obtained is compared with the score of the best prior reflection, and if it is higher, the candidate reflection and its score overwrite the current best reflection in the `short-term memory`. This process repeats until the specified iterations are exhausted, after which the best resulting reflection is stored in the `semantic memory`, following the same logic of a maximum of three reflections as proposed by `Reflexion`. In summary, describing `Retroformer` under `FALAA` clarifies its reflection process—based on the *best-of-n sampler* using a formal description mechanism instead of natural language.

4.3 Comparison

To better illustrate the differences and similarities between `Reflexion` and `Retroformer`, this section compares key aspects of their architectures, focusing specifically on their reflection generation processes and reward assignment strategies.

Reflection Process Both `Reflexion` and `Retroformer` generate reflections only after failing to solve the assigned task. However, the mechanisms underlying the reflection process differ significantly. `Reflexion` generates a single reflection by constructing a prompt and using its LLM to produce feedback (see Figure 3). `Retroformer`, on the other hand, extends this process by introducing the *best-of-n-sampler* method (see Figure 4). `Retroformer`: multiple candidate reflections are generated through a similar prompting mechanism, and a trained reward model is then used to select the best reflection for subsequent use. This enhancement shows how `Retroformer` builds upon `Reflexion` by introducing a more complex reflection cycle that involves additional components and a selection phase.

Reward Assignment In `Reflexion`, rewards are assigned by the `Evaluator` based on the overall outcome of the agent's final trajectory. In contrast, `Retroformer`

assigns rewards incrementally after each action-observation pair generated by the agent. This difference highlights that `Retroformer` implements a more fine-grained evaluation strategy, assessing the agent's performance at each step rather than only at the end of the task, as in `Reflexion`.

5 Conclusion

In this paper, we addressed the lack of a standardized structure and a methodology for describing the architectures of LLM-based agents. Although numerous works have introduced new agent designs and paradigms, these often focus primarily on high-level concepts without exhaustively detailing essential components or behaviors, thus causing ambiguities and making comprehensive understanding, implementation, and comparison difficult. Such challenges lead to agents that do not necessarily meet every aspect required for an unambiguous definition, as seen in `Reflexion` or `Retroformer`, where certain roles and memory structures remain insufficiently specified.

`FALAA` (*Framework for the Abstraction of Language Agent Architectures*) was proposed to mitigate these limitations by offering a general structure belong with a dual-level specification approach. At its core, `FALAA` defines a minimal set of key components (`Planner`, `Executor`, `Evaluator`, `Reflector`, `Memory`, and `Environment`) and a concise action taxonomy, ensuring that every relevant aspect of LLM-based agents can be modeled consistently. On a conceptual level, UML class and sequence diagrams clarify the principal relationships and interactions among these components, while the formal specification level employs OCL to eliminate ambiguities around behaviors, invariants, preconditions, and postconditions.

From the application of this framework, several *insights* were noted:

- *Unnecessary ambiguities revealed and resolved:* By describing agents such as `Reflexion` and `Retroformer` under `FALAA`, it became clear how ambiguities in natural language descriptions (e.g., mixed terminologies, composition of memories or step by step reflection process) can be systematically resolved. This clarity ensures that future agent designs are easier to interpret, compare, and extend.
- *Identification of overlooked design aspects:* The formal specification process highlighted design questions not originally addressed in the literature, such as how many and which past reflections to preserve (`Reflexion` imposes a limit of three).
- *Natural integration across diverse agent paradigms:* `FALAA` does not forcibly restructure existing architectures but instead maps their essential components into a shared nomenclature (e.g. the integration of `Actor` component on `Reflexion` and `Retroformer` architectures). This shows that many of the behaviors in LLM-based agents are fundamentally similar, allowing for more coherent cross-agent comparisons.

Additionally, using `FALAA` *facilitates* the development and documentation process of LLM-based agents. By providing a clear set of responsibilities for each component and specifying how they interact, authors are steered toward more precise and transparent explanations of their systems. This enables researchers to systematically analyze which part of an agent's performance could be improved or replaced, as well as to compare existing architectures on a like-for-like basis, thus imposing a more consistent convention in the terminology of this emerging domain.

After analyzing some agents under `FALAA`, we have identified various opportunities for *future work*:

- **Multi-Agent Systems (MAS):** Extending this framework to scenarios where multiple LLM-based agents interact could yield significant benefits, particularly in tasks requiring collaboration or negotiation. Future work could include defining additional `FALAA` components and interaction protocols specific to MAS.
- **Improved Reflection Mechanisms:** Both `Reflexion` and `Retroformer` highlight reflection processes but do not reuse older reflections in subsequent task attempts. Enhancing the storage and retrieval of such knowledge in `Semantic memory` (or other new memory abstractions) could allow agents to leverage insights gained from past tasks more effectively.

Overall, the proposal of `FALAA` aims to streamline the design, specification, and understanding of language-agent architectures by introducing a clear structure and methodology. The framework not only provide a higher degree of clarity and consistency but also leaves room for further innovation in agent design. We anticipate that `FALAA`, by fostering uniformity and comparability across heterogeneous research efforts, could help shape the next generation of LLM-based agents, promoting both incremental improvements and further innovation in this rapidly evolving field.

Acknowledgments. This work was supported by Millennium Institute for Foundational Research on Data (IMFD), ANID Millennium Science Initiative Program Code ICN17_002 and the National Center for Artificial Intelligence CENIA FB210017, Basal ANID.

References

1. Cheng, Y., Zhang, C., Zhang, Z., Meng, X., Hong, S., Li, W., Wang, Z., Wang, Z., Yin, F., Zhao, J., He, X.: Exploring large language model based intelligent agents: Definitions, methods, and prospects (2024), https://arxiv.org/abs/2401.03428
2. Crouse, M., Abdelaziz, I., Astudillo, R., Basu, K., Dan, S., Kumaravel, S., Fokoue, A., Kapanipathi, P., Roukos, S., Lastras, L.: Formally specifying the high-level behavior of llm-based agents (2024), https://arxiv.org/abs/2310.08535
3. Franklin, S., Graesser, A.: Is it an agent, or just a program?: A taxonomy for autonomous agents. In: Müller, J.P., Wooldridge, M.J., Jennings, N.R. (eds.) Intelligent Agents III Agent Theories, Architectures, and Languages. pp. 21–35. Springer Berlin Heidelberg, Berlin, Heidelberg (1997)
4. Liu, Z., Yao, W., Zhang, J., Xue, L., Heinecke, S., Murthy, R., Feng, Y., Chen, Z., Niebles, J.C., Arpit, D., Xu, R., Mui, P., Wang, H., Xiong, C., Savarese, S.: Bolaa: Benchmarking and orchestrating llm-augmented autonomous agents (2023), https://arxiv.org/abs/2308.05960
5. Object Management Group: Object constraint language (ocl), version 2.4. Specification formal/2014-02-03, Object Management Group (February 2014), https://www.omg.org/spec/OCL/2.4/PDF
6. OpenAI: Gpt-4 (2023), https://openai.com/research/gpt-4, accessed: 2024-08-23
7. Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C.L., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., Schulman, J., Hilton, J., Kelton, F., Miller, L., Simens, M., Askell, A., Welinder, P., Christiano, P., Leike, J., Lowe, R.: Training language models to follow instructions with human feedback (2022), https://arxiv.org/abs/2203.02155
8. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach. Prentice Hall, 3 edn. (2010)
9. Russell, S.J., Norvig, P.: Artificial intelligence: a modern approach. Pearson (2016)
10. Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., Yao, S.: Reflexion: language agents with verbal reinforcement learning. In: Oh, A., Neumann, T., Globerson, A., Saenko, K., Hardt, M., Levine, S. (eds.) Advances in Neural Information Processing Systems. vol. 36, pp. 8634–8652. Curran Associates, Inc. (2023), https://proceedings.neurips.cc/paper_files/paper/2023/file/1b44b878bb782e6954cd888628510e90-Paper-Conference.pdf
11. Sumers, T.R., Yao, S., Narasimhan, K., Griffiths, T.L.: Cognitive architectures for language agents (2024), https://arxiv.org/abs/2309.02427
12. Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez, A., Joulin, A., Grave, E., Lample, G.: Llama: Open and efficient foundation language models (2023), https://arxiv.org/abs/2302.13971
13. Wang, G., Xie, Y., Jiang, Y., Mandlekar, A., Xiao, C., Zhu, Y., Fan, L., Anandkumar, A.: Voyager: An open-ended embodied agent with large language models (2023)
14. Wang, L., Ma, C., Feng, X., Zhang, Z., Yang, H., Zhang, J., Chen, Z., Tang, J., Chen, X., Lin, Y., Zhao, W.X., Wei, Z., Wen, J.: A survey on large language model based autonomous agents. Frontiers of Computer Science **18**(6) (Mar 2024). https://doi.org/10.1007/s11704-024-40231-1, http://dx.doi.org/10.1007/s11704-024-40231-1

15. Weng, L.: Llm-powered autonomous agents. https://lilianweng.github.io/posts/2023-06-23-agent/ (June 2023), lilianweng.github.io
16. Xi, Z., Chen, W., Guo, X., He, W., Ding, Y., Hong, B., Zhang, M., Wang, J., Jin, S., Zhou, E., Zheng, R., Fan, X., Wang, X., Xiong, L., Zhou, Y., Wang, W., Jiang, C., Zou, Y., Liu, X., Yin, Z., Dou, S., Weng, R., Cheng, W., Zhang, Q., Qin, W., Zheng, Y., Qiu, X., Huang, X., Gui, T.: The rise and potential of large language model based agents: A survey (2023), https://arxiv.org/abs/2309.07864
17. Yao, W., Heinecke, S., Niebles, J.C., Liu, Z., Feng, Y., Xue, L., Murthy, R., Chen, Z., Zhang, J., Arpit, D., Xu, R., Mui, P., Wang, H., Xiong, C., Savarese, S.: Retroformer: Retrospective large language agents with policy gradient optimization (2024)
18. Zhang, Z., Yao, Y., Zhang, A., Tang, X., Ma, X., He, Z., Wang, Y., Gerstein, M., Wang, R., Liu, G., Zhao, H.: Igniting language intelligence: The hitchhiker's guide from chain-of-thought reasoning to language agents (2023)