A Reflexion

A.1 Ilustrative Example.

With the goal of demonstrating how a language agent is described under the standardized structure and the description methodology proposed by `FALAA`, the `Reflexion` architecture [3] will be presented below. `Reflexion` is an agent that focuses on providing LLM-based agents with *linguistic feedback* after a task failure. The failed trajectory generated by solving a given task, is transformed into a textual summary that guides subsequent attempts. To implement this approach, [3] define three main components: an *Actor* (responsible for action generation), an *Evaluator* (in charge of assigning rewards or ratings to the trajectory), and a *Self-reflection* module (responsible for generating feedback to correct errors). Additionally, the authors define short-term memory (referred to as *trajectory*) for storing recent actions and observations, and long-term memory (*Mem*) for preserving reflections.

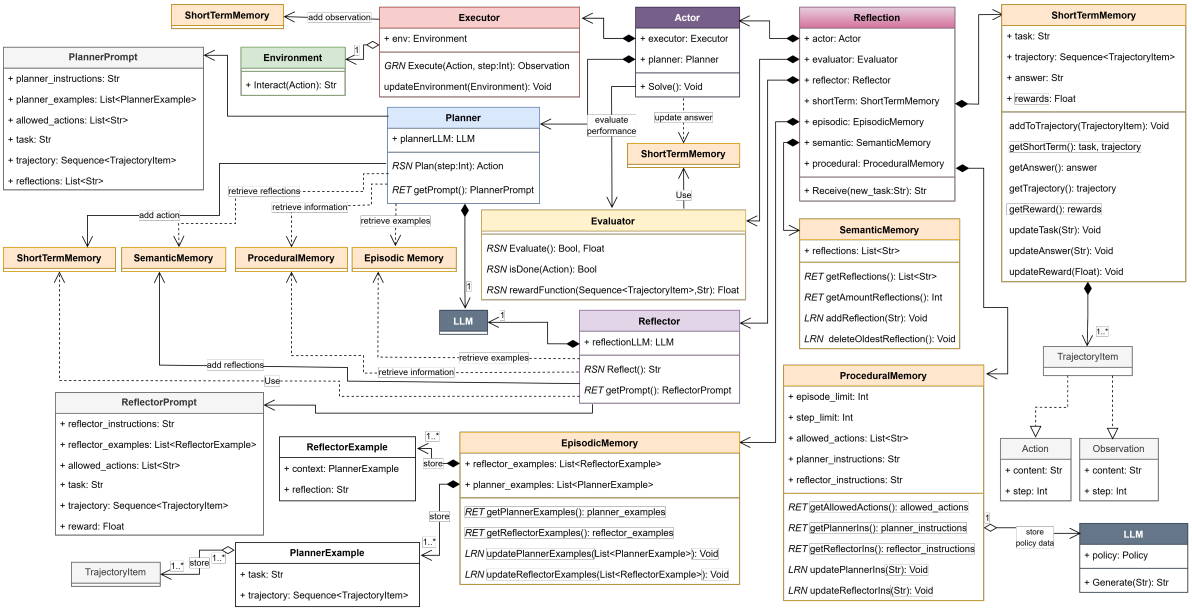

Fig. 1: Class diagram for the `Reflexion` agent using `FALAA` standarized structure.

(1) Conceptual Description Level. Since `Reflexion` is an already established architecture, the first step is to translate its original components into the standardized structure proposed by `FALAA`. Figure 1 shows the class diagram of `Reflexion`, where we establish that both *Evaluator* and *Self-reflection* correspond to the `Evaluator` and `Reflector` components, respectively. In this architecture, the main responsibilities of `Evaluator` are to determine whether the action produced by `Planner` corresponds to the final response (which will be described at the formal specification level) and to assign, using its own reward function, a score to the trajectory generated by the agent after finding a solution to the assigned task. On the other hand, `Reflector`, aside from its responsibility of generating corrective feedback through a *reasoning action*, is responsible for using the *retrieval actions* from memories to construct the prompt for its LLM. We decided to abstract prompts in this description as objects to explicitly define their content.

Regarding memory, long-term memory (*Mem*) is described under the `FALAA`'s structure as a list of reflections stored in the `semantic memory`, due to its nature of storing knowledge and reflections for decision-making. On the other hand, short-term memory (*trajectory*) is stored under the same name and is represented as an attribute of the `short-term memory` component in `FALAA`. Action and Observation objects were defined to represent the content of the trajectory, which will facilitate the description of its behavior at the formalization level. Finally, two objects are established to represent the examples included in the prompts for `Planner` and `Reflector`,

which are stored in the `episodic memory`.

In the original description, *Actor* holds the responsibilities of the standard components `Planner` and `Executor`; therefore, to explicitly allow the possibility of adding additional components, we decided to maintain this component and describe it as one composed of `Planner` and `Executor`, with the responsibility of managing the entire process of searching for solutions to an assigned task.

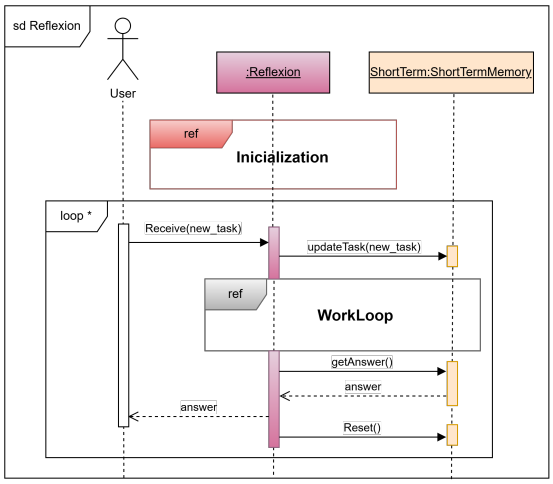

Fig. 2: UML main sequence diagram of the Reflexion LLM-based agent.

Behavior Description. After briefly describing the components of `Reflexion`, `FALAA` proposes to describe the behavior of the components through sequence diagrams. Figure 2 illustrates how, before interacting with the user, the agent's memory initialization process is explicitly carried out (see Figure 3), where instructions and examples are added for the LLMs of `Planner` and `Reflector`. After initialization, an infinite loop begins where the agent receives a new task from the user, which is added to the `short-term memory` and then proceeds with the work cycle—summarized by the fragment **ref** *Work Loop*—(see Figure 4). Upon completion of this process, the generated response is retrieved from the `short-term memory` and returned to the user. It is not explicitly stated what should be done if no response is found, so we assumed that the agent does not execute any further actions. Finally, the `short-term memory` is cleared using its *Reset()* method, and the agent remains on standby for a new task.

The work cycle of `Reflexion` (see figure 4) consists of a loop that runs from 1 to the maximum value *episode_limit*, which indicates the number of failed responses allowed before terminating the cycle. In each iteration, `Reflexion` calls the *Solve()* method of the `Actor` (see Figure 5), which is responsible for solving the given task by generating a response.

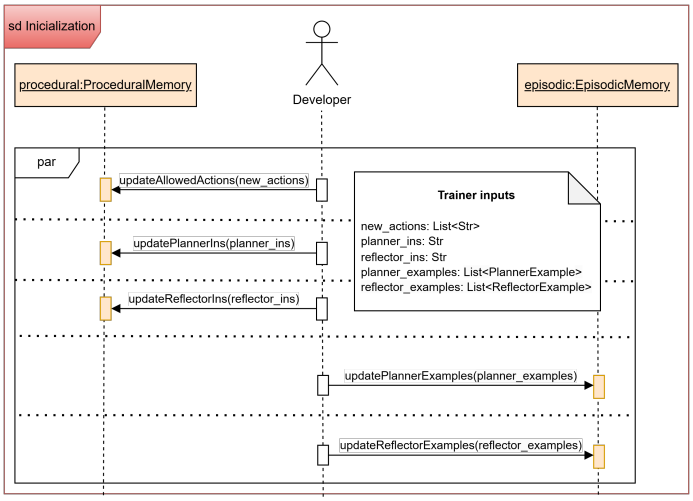

Fig. 3: UML sequence diagram of the initialization process of a `Reflexion` agent, illustrating how a developer initializes the `episodic` and `procedural` memories with examples, allowed actions, and instructions. A comment within the diagram specifies the names and types of the inputs for each method, facilitating diagram readability. This information is crucial for the subsequent processes of the `Planner` and `Reflector` components.

This response will be evaluated by `Evaluator` through its *Evaluate()* method. Internally, this method uses its reward function to assign a score to the response based on the current trajectory used. After storing the reward in the `short-term memory`, a boolean value is returned indicating whether the response is correct or not.

If the response is incorrect, the process enters an **opt** fragment where `Reflexion` executes the *Reflect()* method of the `Reflector` (see Figure 8), which generates a reflection based on the failed response. Finally, a new attempt is initiated in the work cycle, trying to solve the task again but now with an added reflection in the `semantic memory`, which will be used by the `Planner` to generate new actions.

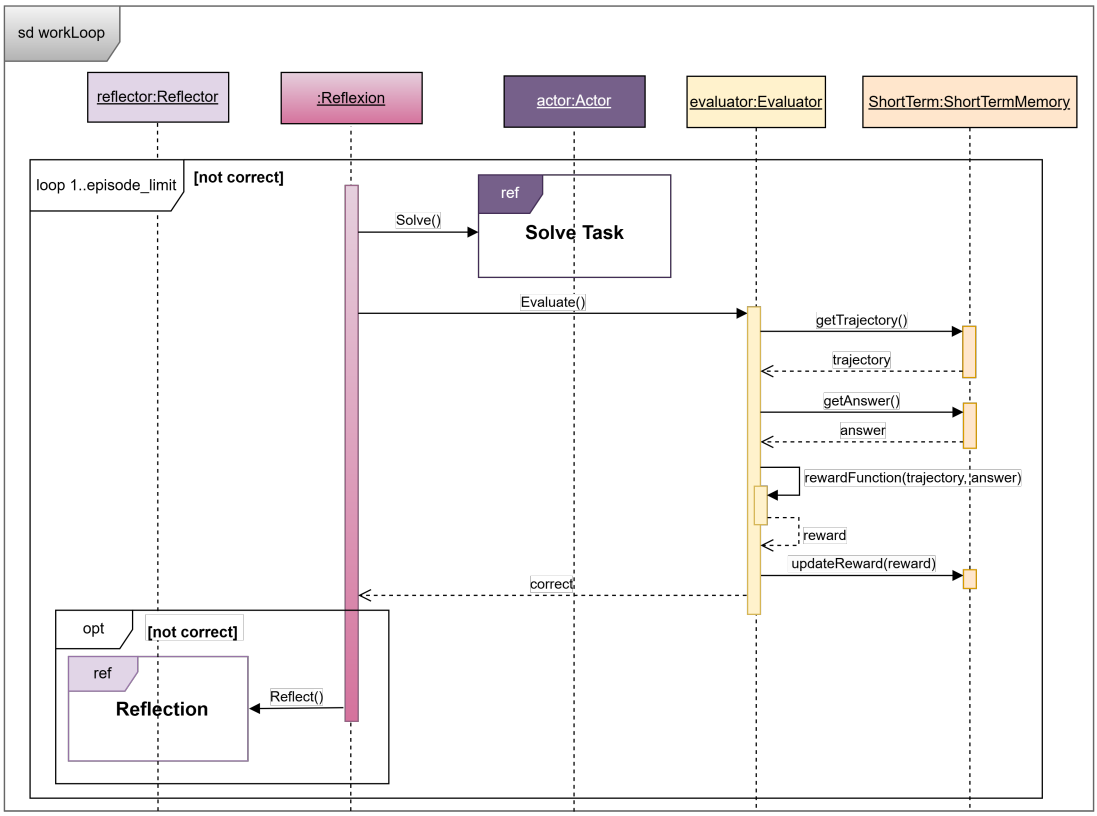

Fig. 4: UML sequence diagram of the work loop process of the Reflexion agent to solve a given task.

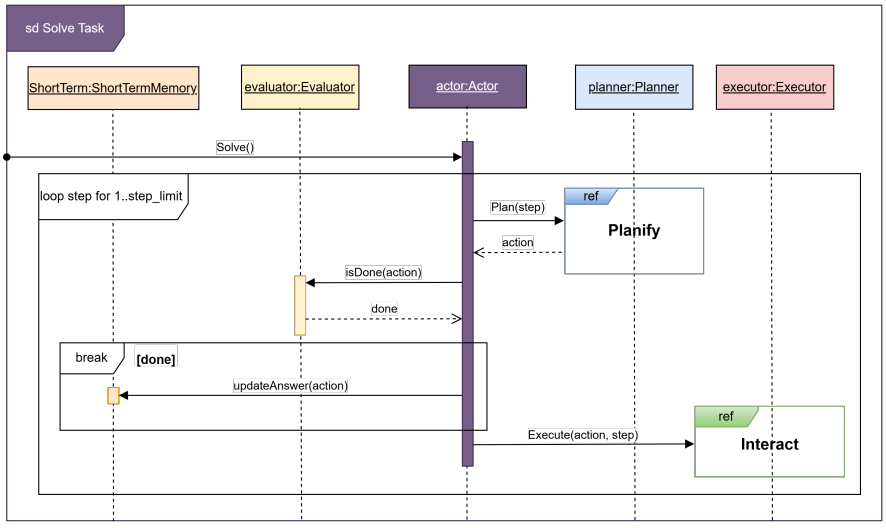

Fig. 5: UML sequence diagram of the Actor component of the Reflexion agent.

Figure 5 presents the process of solving a task by the `Actor` component of `Reflexion`. This process is kept as general as possible since [3] indicates that the `Actor` can be part of another agent. The entire process is encapsulated in a step cycle that ranges from 1 to *step_limit*, which indicates the maximum number of actions and observations that can be performed to solve the task.

In each step, `Planner` is used to provide an action to solve the task (see Figure 6). In this diagram, We present a general flow of planning process of the `Planner` component of a `Reflexion` agent because [3] suggests that the entire process of the `Actor`, and consequently that of `Planner` and `Executor`, can be replaced by other agents.

When the `Planner`'s *Plan* method is called, it receives the current step value, which is stored in each action, serving as a temporary guide for the components that read the trajectory. After creating the prompt for its LLM and retrieving relevant information from the memories (retrieval planner info fragment in figure 9), the `Planner` uses its LLM to generate an action in string format. This action, along with the received step value, is used to create an Action object. The new action is then added to the trajectory in the `Short-term` memory and returned.

This action is evaluated by `Evaluator` to determine if the action generated contain the keyword "[answer]". If it is not a final response, the generated action is executed by the `Executor`, showed in the UML sequence fragment in figure 7. This fragment details how the `Executor` receives the action, executes it in the environment, and obtains the observation generated by the action, which is then stored in the trajectory of the `short-term memory`.

Figure 8 describe the behavior of the main contribution of [3]: The self-reflection process. In this process, we describe how, after obtaining the necessary prompt (retrieval reflector info fragment in figure 10), `Reflector` uses its LLM to generate a reflection on the possible cause of why the task could not be solved in the previous trajectory. This reflection is added to the `semantic memory` to be used by `Planner` in generating new actions.

Before adding the new reflection, it is always verified that there are no more than 3 reflections in the semantic memory, removing the oldest one if this condition is met. This invariant will be formally specified in the formal specification section.

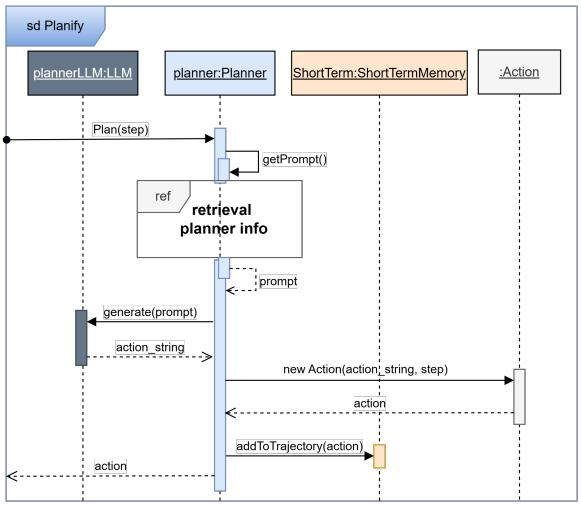

Fig. 6: Planning process of `Reflexion's Planner` component.

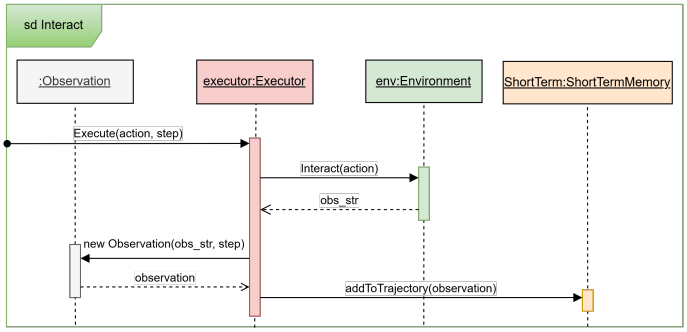

Fig. 7: UML sequence diagram of the interaction process of the Reflexion agent.

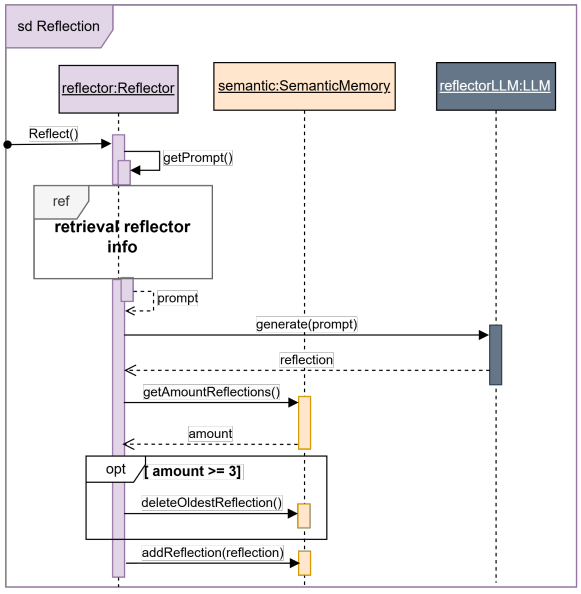

Fig. 8: UML sequence diagram of the reflection process of the Reflexion agent.

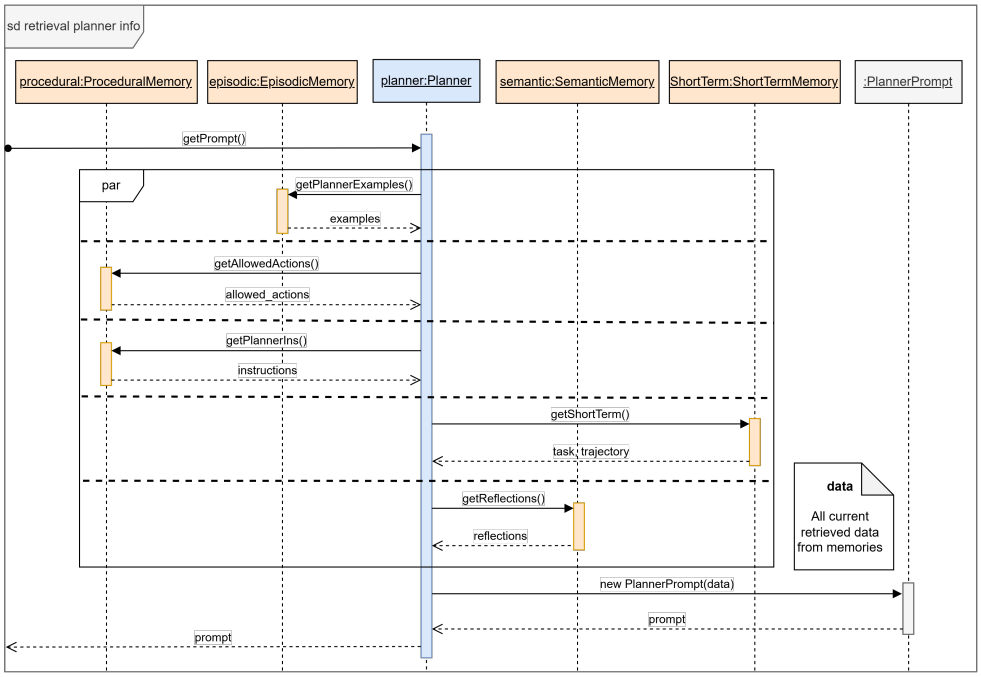

Fig. 9: UML sequence diagram of the information retrieval process for `Planner` to generate a prompt. In this flow, `Planner` uses internal actions of type `retrieval`, represented by the *get* methods of the `episodic`, `procedural`, and `semantic` memories, from which examples are obtained along with the allowed actions and instructions. These are condensed into the *PlannerPrompt* object, aiming to guide the behavior of the `Planner` LLM to generate actions.

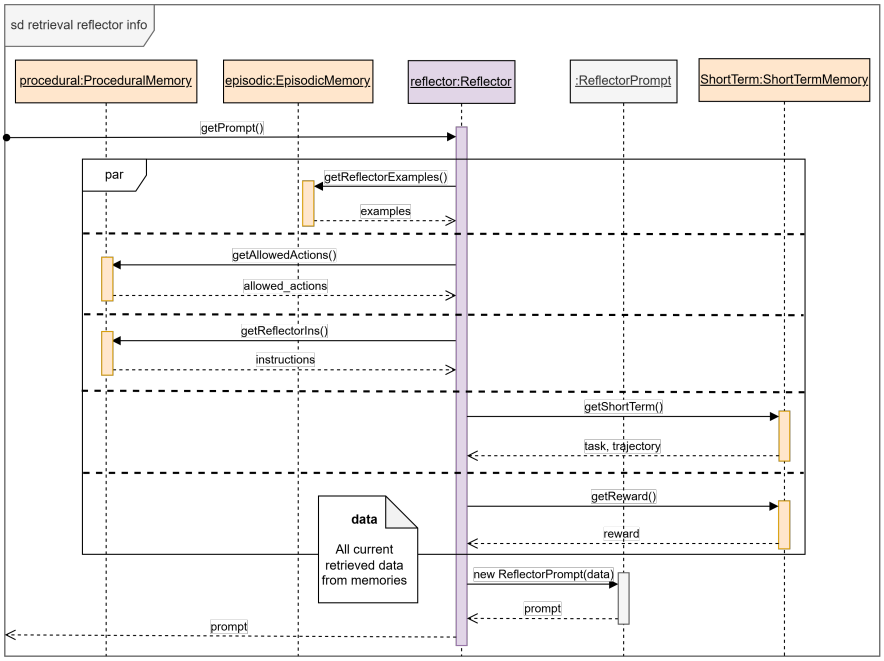

Fig. 10: UML sequence diagram of the information retrieval process for `Reflector` to generate a prompt. In this flow, `Reflector` uses internal actions of type `retrieval`, represented by the *get* methods of the `episodic` and `procedural` memories, from which examples for the LLM and the allowed actions along with the instructions to guide the LLM behavior are obtained. These, together with the task and current trajectory retrieved from the `short-term memory`, are used to generate the prompt.

(2) Formal Specification Level In the `Reflexion` architecture, we have identified six behaviors which, to avoid ambiguities, need to be formally specified. These behaviors are:

1. The *Reset* method of the `short-term memory`, which must clear its attributes in a specific format.
2. The *isDone(Action)* method of the `Evaluator`, which must return true if the received action contains the string "[answer]".
3. The limit of 3 reflections that `Reflexion` allows to be stored in `Semantic memory`.
4. The respective precondition of the `Semantic memory` *addReflection* method, which allows adding a new reflection only if there are less than 3 stored reflections.
5. The invariant of the trajectory in the `short-term memory`, which must follow an alternating action-observation pattern.
6. Consequently, the preconditions and postconditions that the *addToTrajectory* method of the `short-term memory` must satisfy to preserve this invariant.

For behavior 1, the specification 1 addresses the *Reset()* method of the `short-term memory`. It explicitly defines how the values of the trajectory, the reward from the previous episode, and other attributes should be reset.

For behavior 2, the OCL specification 2 is defined, which states that the *isDone(Action)* method of the `Evaluator` must return true if the content of the received action contains the substring "[answer]". Otherwise, it must return false.

Specification 3 establishes an invariant which ensures that the reflections stored in `Semantic memory` are less than three. On the same hand, specification 4 formalizes the behavior 4.

Regarding trajectory behavior, the invariant named *TrajectoryAlternation* is defined, as shown in the OCL specification 5. According to this invariant, it must always be satisfied that either the trajectory of the `short-term memory` is empty or, if not, the first element of the trajectory must be of type `Action`, and for each element in the trajectory, if it is of type `Action`, the next one must be of type `Observation`.

After defining the trajectory invariant, the OCL specification 6 addresses the *addToTrajectory* method, which adds elements to the trajectory of the `short-term memory`. A precondition and two postconditions are defined.

context `ShortTermMemory::Reset(): OclVoid`
post:
self.task = "" (1)
post:
self.answer = ""
post:
self.trajectory → isEmpty()
post:
self.rewards = 0

---

context `Evaluator::isDone(action : Action): Bool`
post: (2)
let actionContent: String = action **in**
result = actionContent.indexOf("[answer]") > 0

---

context `SemanticMemory` **inv** MaxThreeReflections :
self.reflections → size() ≤ 3 (3)

---

context `SemanticMemory` :: addReflection(
newReflection : Reflection): OclVoid
pre :
self.reflections → size() < 3 (4)

The precondition, named *MaintainAlternation*, establishes that when calling the method, if the trajectory is empty, the new element to be added must be of type `Action`. If the trajectory already contains elements, it verifies that the last element corresponds to the new element to be added, maintaining the action-observation sequence. The postconditions *IncreasedSize* and *NewElementIsLast* state that after adding a new element to the trajectory, its size must increase, and the new element must be the last one in the trajectory, respectively.

```
context ShortTermMemory inv TrajectoryAlternation :
  self.trajectory → isEmpty()
  or
    let size: Integer = self.trajectory → size() in
    Sequence{1..size} → forAll(i: Integer |                    (5)
      if i.mod(2) = 1 then
        self.trajectory → at(i).oclIsTypeOf(Action)
      else
        self.trajectory → at(i).oclIsTypeOf(Observation)
       endif
    )
```

```
context ShortTermMemory :: addToTrajectory(
              newItem : TrajectoryItem): OclVoid
pre MaintainAlternation:
  if self.trajectory → isEmpty() then
    newItem.oclIsTypeOf(Action)
  else let lastElement = self.trajectory → last() in
    if lastElement.oclIsTypeOf(Action) then
      newItem.oclIsTypeOf(Observation)                         (6)
    else if lastElement.oclIsTypeOf(Observation) then
      newItem.oclIsTypeOf(Action)
  endif
post IncreasedSize:
  self.trajectory → size() = self.trajectory@pre → size() + 1
post NewElementIsLast:
  self.trajectory → last() = newItem
```

B Retroformer

B.1 Pseudo Code

Figure 11 shows the training process based on RLHF, which was ultimately used as a guide to decide whether `Retroformer` would be trained for one or multiple environments.

Algorithm 1 **`Retroformer`** with Policy Gradient Optimization

Initialize TEXT-DAVINCI-003 as the Retrospective model with LONGCHAT-16K. Set the maximum trials for rollouts as $N = 3$. The temperature used for sampling $t_s = 0.9$.
Step 1: Offline Data Collection. Collect multiple rollouts for each environments k ($k = 1, \cdots, K$) for the tasks in the training sets and save as D_{RL}.
for episode t = 1, . . . , N **do**
  for source domain k = 1, . . . , K **do**
    Receive trajectory $[s_{k,i,\tau}, a_{k,i,\tau}, r_{k,i,\tau}]_{\tau=1}^{T}$ and episodic returns $G_{k,i}$ for task i.
    for unsuccessful tasks j **do**
      Randomly sample a pair of reflection responses $(y_{k,j}^{(1)}, y_{k,j}^{(2)})$ with Retrospective LM temperature set to t_s, with the same instruction prompt defined in Eq. (4).
      Roll out the next episode with $y_{k,j}$, and receive the episodic returns $(G_{k,i+1}^{(1)}, G_{k,i+1}^{(2)})$.
      Compute reflection response rating by $r(x_{k,i}, y_{k,i}) \triangleq G_{k,i+1} - G_{k,i}$ in Eq. (5).
      Label the response with higher ratings as the accepted response while the lower response is labeled as the rejected response.
    end for
  end for
end for
Step 2. Reward Model Learning. Use the REWARDTRAINER in TRL to train a model for classifying accepted and rejected responses given instructions.
Step 3: Policy Gradient Finetuning. Plug-in the trained reward model and use the PPOTRAINER in TRL to finetune the Retrospective model for generating reflection responses with higher ratings.

Fig. 11: Diagram outlining the RLHF-based training process, consisting of three steps: Obtaining training data; Training a reward model with Supervised Learning; and fine-tuning the *retrospective model* through response ranking using PPO [extracted from [4]].

B.2 Architecture Description using FALAA

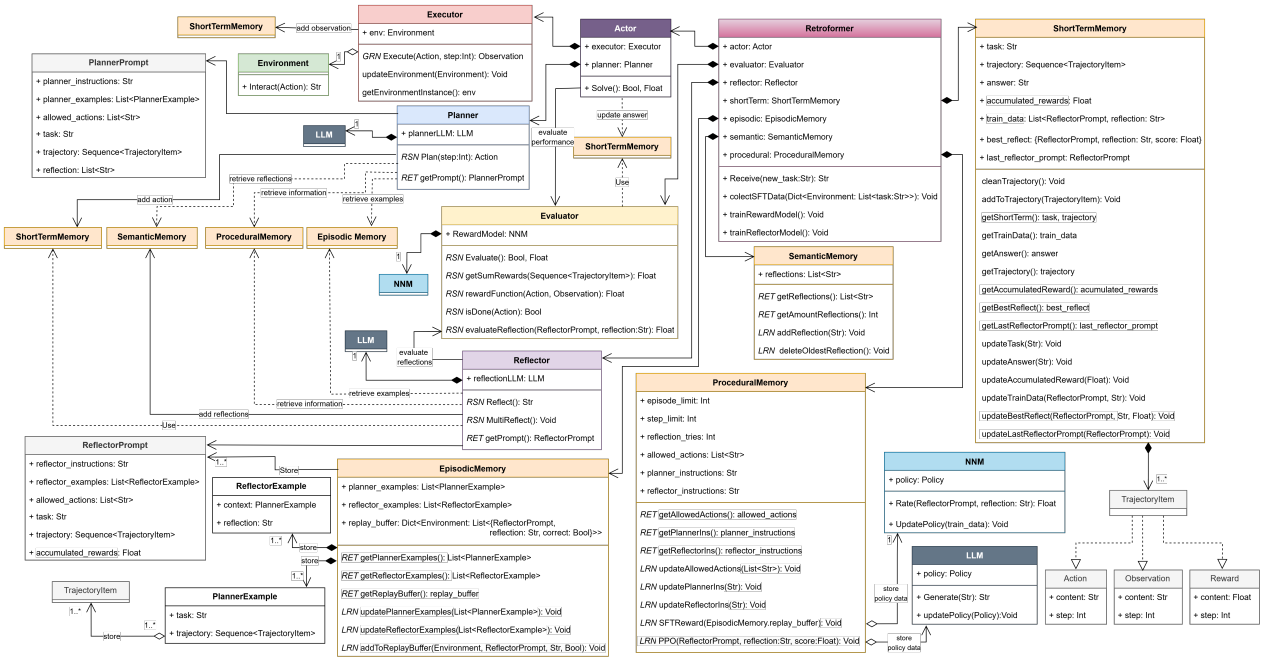

Fig. 12: UML class diagram of a `Retroformer` agent.

(1) Conceptual Description Level Components Description. `Retroformer` [4] originally proposes three fundamental components: an *Actor*, a *Retrospective model* and a memory module (short-term and long-term memory, as well as a *replay buffer*). Along with this components, a reward model is trained and used. Figure 12 shows the UML diagram exemplifying how these components are integrated under the standard structure proposed by `FALAA`. Each component is presented below, providing the original definition and its respective standardized adaptation:

1. **Actor**
   - *Original Definition*: The *Actor* is described, like in `Reflexion`, as the component responsible for solving the assigned task, using an LLM to generate and execute actions in an environment.
   - *Adaptation in* ***FALAA***:
     - Under the standardized structure, it is defined as an additional component, consisting of two main subcomponents: the `Planner`, which is responsible for planning actions, and the `Executor`, which executes them.
     - The original concept of "solving the task" is maintained, but adapted to fit within the `FALAA` structure for better integration and compatibility with other agents.
2. ***Retrospective model***
   - *Original Definition*: It consists of a smaller LLM compared to the *Actor* and is responsible for producing reflections (*feedback*) when the *Actor* fails in the task.
   - *Adaptation in* ***FALAA***:
     - It is associated with the `Reflector` component, whose function is to generate reflections and insights on failures or possible improvements in the `Actor`'s strategy.
3. **Reward Model**
   - *Original Definition*: Specifically trained by `Retroformer` to evaluate the quality of reflections generated by the *Retrospective model*.
   - *Adaptation in* ***FALAA***:
     - It is designated as an independent object within the architecture, similar to LLMs, dedicated to evaluation and reward assignment.

4. **Memory Module**
   - *Original Definition*:
     - *Short-term memory*: Stores the trajectory (actions, observations, rewards) generated by the *Actor*.
     - *Long-term memory*: Saves reflections produced by the *Retrospective model*.
     - *Replay buffer*: Specialized memory for storing triplets of `prompts`, reflections, and the accumulated reward of the episode.
   - *Adaptation in* `FALAA`:
     - `Short-term memory`: Under the standard structure, it stores a series of attributes, including the trajectory generated by the `Actor` and the accumulated rewards of that trajectory, as well as attributes aimed at the training process of `Retroformer`.
     - `Long-term memory for reflections`: Reflections are stored in the `semantic memory` due to its nature of storing knowledge.
     - `Replay buffer`: This particular memory is established as a dictionary that stores a triplet of prompts, reflections, and their classification as correct or incorrect for various environments. It is stored in the `episodic memory` due to its nature of storing past experiences.

`Retroformer` utilizes all structures proposed by `FALAA`, with the `procedural memory` playing a key role due to the architecture's training processes. This memory in the `Retroformer` architecture it can execute *learning actions* through methods that implicitly update stored information, modifying the LLM and neural network-based reward model policies.

The training process also influences the `short-term memory`, which stores various information such as *train_ data*. The trajectory stored in the `short-term memory` also undergoes modifications, adopting a sequence that now includes the elements `Action`, `Observation`, and `Reward`, reflecting the new architectural needs for a more structured and efficient process.

We define several Objects representing prompts and examples used in the creation of actions and reflections by `Planner` and `Reflector` respectively. This is to make their composition explicit, as in implementations they are often text strings or token lists.

The `Evaluator` component plays a crucial role within the architecture. Since it is responsible for generating evaluations and thus rewards, it integrates the reward model dedicated to evaluating the quality of reflections. `Evaluator` offers various functional methods, including the evaluation of an action-observation pair through a reward function, calculating the sum of rewards in a given trajectory, determining whether an action corresponds to a final response, and assessing the quality of a reflection.

In general, the architecture of `Retroformer` partially retains the structure of `Reflexion`, but introduces modifications and new functionalities, such as the use of rewards for each step of the trajectory or new attributes in the `short-term memory` used in the training processes of the reward model and retrospective model.

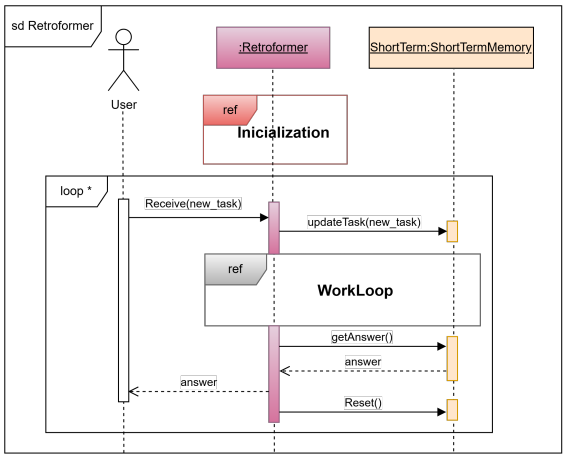

Fig. 13: UML main sequence diagram of the Retroformer LLM-based agent.

Behavior Description. Figure 13 shows the main UML sequence diagram of a `Retroformer` agent. Before interacting with the user, `Retroformer` exists in a certain environment, so its memories must be initialized (this fragment is identical to `Reflexion` in Figure 3). Once initialized, an infinite loop begins where the agent receives a new task from the user, which is added to the `short-term memory` before proceeding with the work cycle (condensed under the *ref* fragment referencing Figure 14). After completing this process, the generated response is retrieved from the `short-term memory` and returned to the user. Similar to `Reflexion`, it is not explicitly described what should happen if the response is not found, so we assume that the agent does not perform any additional

actions. Finally, the `short-term memory` is cleared using its *Reset()* method, leaving the agent ready to receive a new task.

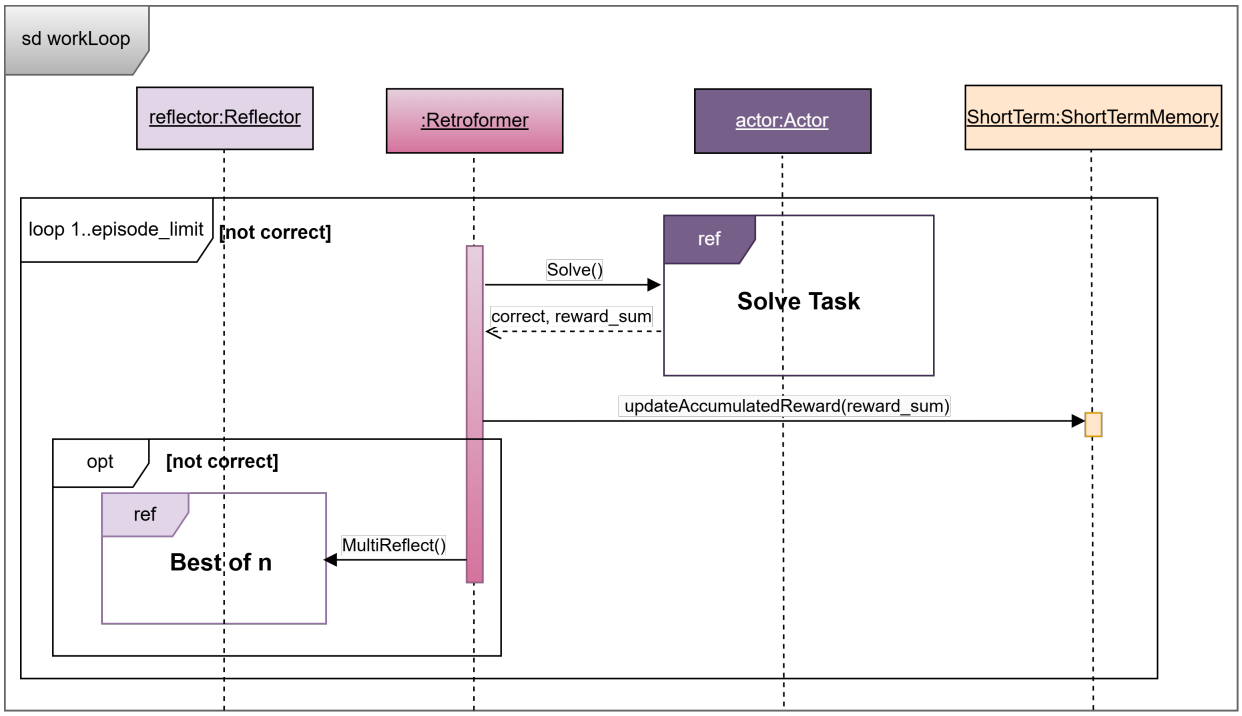

Fig. 14: UML sequence diagram of the work cycle of a `Retroformer` agent.

Regarding the referenced work cycle (see Figure 14), it is observed that the `Retroformer` agent maintains a work cycle similar to that of `Reflexion`, consisting of a loop running from 1 to the maximum value *episode_limit*, which indicates the number of failed responses allowed before ending the cycle. The loop continues as long as the response generated in the previous iteration is not correct. In each iteration, the `Actor` attempts to solve the task (see Figure 15), returning the sum of the accumulated rewards in its trajectory along with an indication of whether the task was successfully resolved. If the task is not successfully resolved, the process of generating reflections is executed, introducing a new flow (see Figure 17). Finally, a new iteration begins.

The task resolution process of the `Actor`, as shown in Figure 15, remains as general as possible, since `Retroformer` allows replacing the `Actor` with components from other agents. The task resolution flow is based on two parts.

The first part is a loop with a maximum of *step_limit* steps, where in each step, `Planner` is asked to generate an action (the planning and corresponding information retrieval process for prompt creation is the same as in `Reflexion`, see Figures 6 and 9, respectively). The generated action is then evaluated to determine if it is the final response, which would end the loop. If not, `Executor` executes the action, receiving an observation that is added to the trajectory in the `short-term memory` (the process is the same as in `Reflexion`, see Figure 7). Then, the action-observation pair is evaluated using the reward function of `Evaluator`, returning a score for this step, which is converted into a *Reward* object and added to the `short-term memory`.

The second part occurs when the loop ends, either by reaching the step limit or by finding the correct response. The current episode is then evaluated, where the accumulated rewards from the current trajectory are obtained, and it is determined whether the response is correct or not, returning these two values.

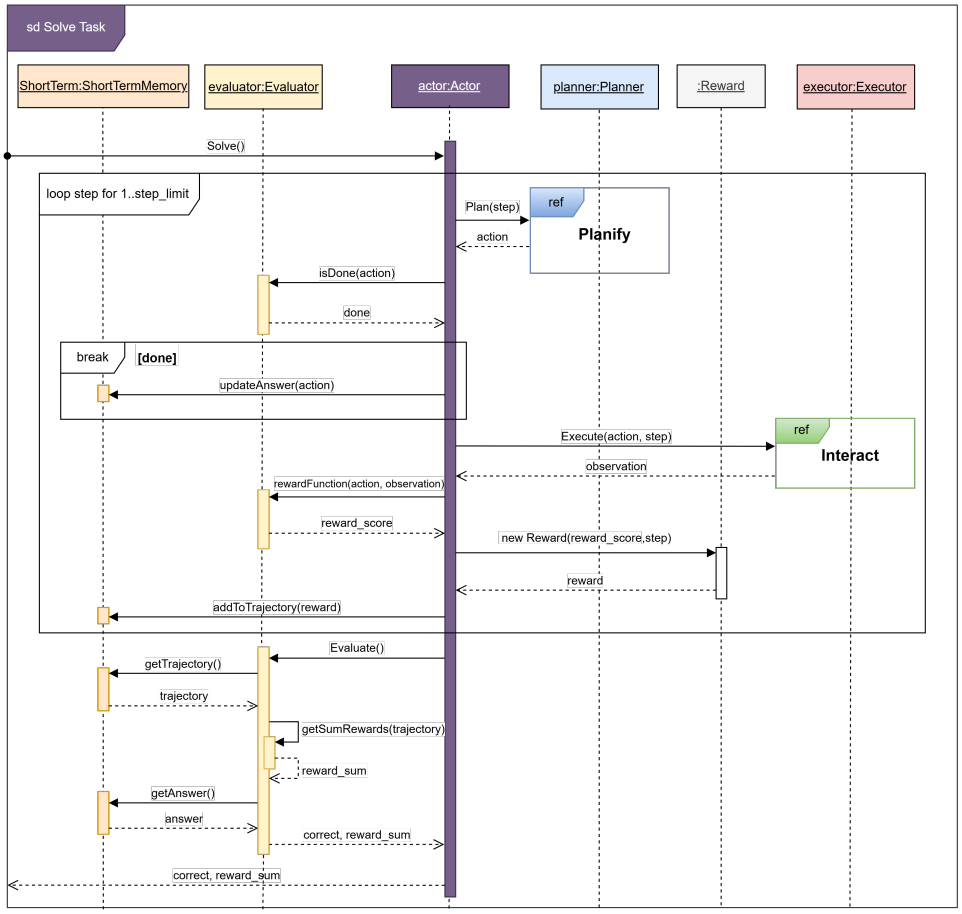

Fig. 15: UML sequence diagram of the task resolution process by the `Actor` component of a `Retroformer` agent.

Figure 17 shows the process of selecting the best reflection from the `Reflector` component of a `Retroformer` agent, which is the main innovation of this architecture. It illustrates how `Reflector` generates *reflection_tries* reflections in a loop. In each step of this loop, a reflection is generated using the *reflect()* method (see Figure 16), where a prompt composed of information from short-term and long-term memories (see Figure 18) is used to generate a reflection, which is then returned.

Once the reflection is generated, it is evaluated using the `Evaluator` component, which acts as an intermediary. Using the prompt and the generated reflection, `Evaluator` applies its pretrained neural network model to assess the quality of the reflection. If the reflection is better than the best one stored so far, it is saved as the best reflection in the `short-term memory`. At the end of the cycle, the best reflection is stored in the `semantic memory`, ensuring that the storage limit is not exceeded.

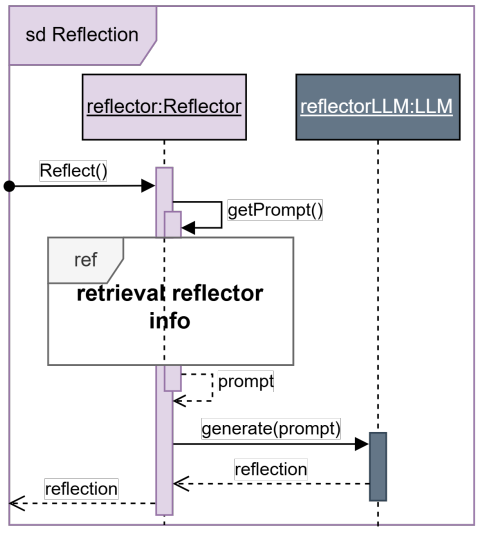

Fig. 16: UML sequence diagram of the reflection process of the `Reflector` component of a `Retroformer` agent.

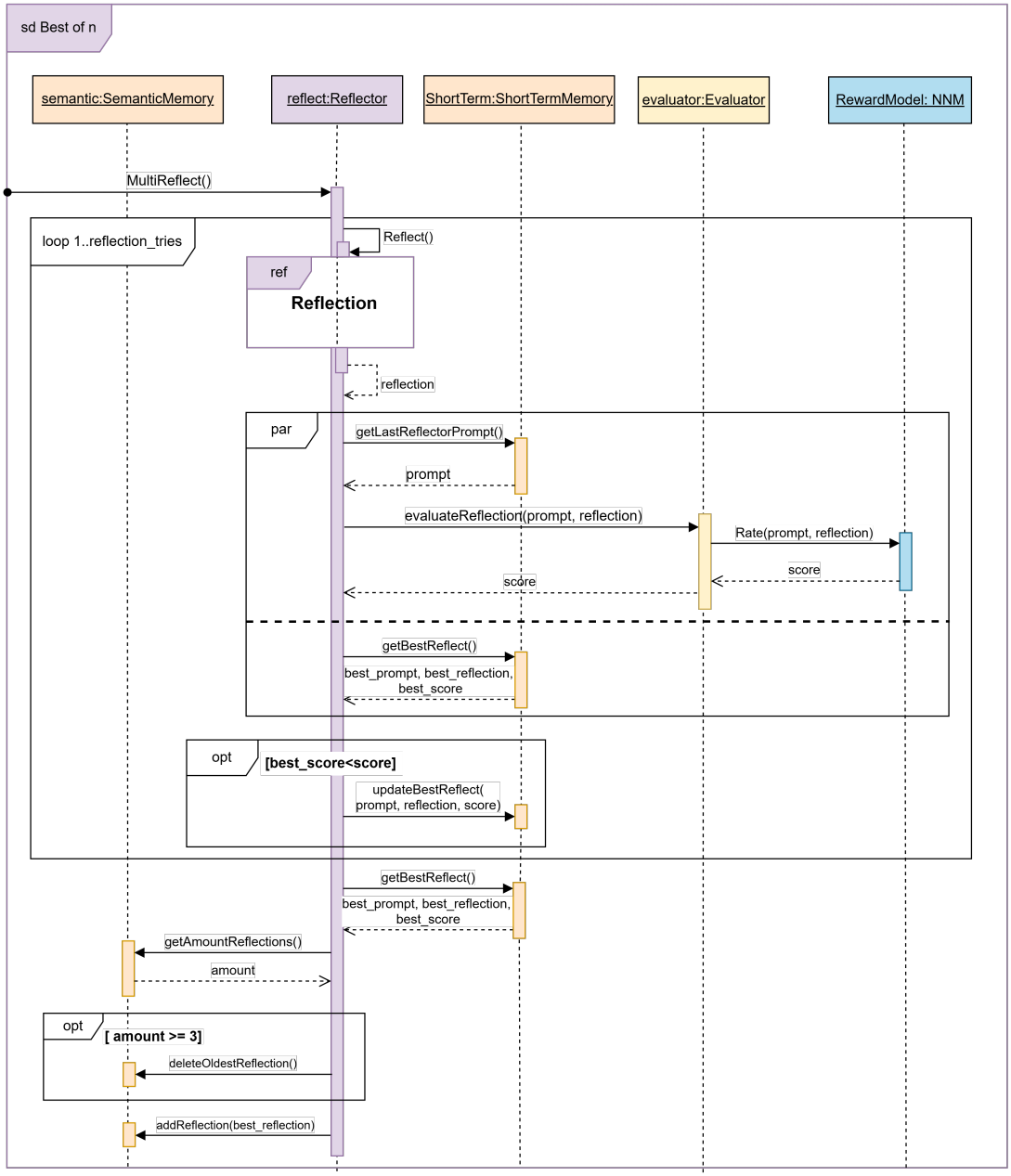

Fig. 17: UML sequence diagram of the best reflection selection process for the `Reflector` component of a `Retroformer` agent.

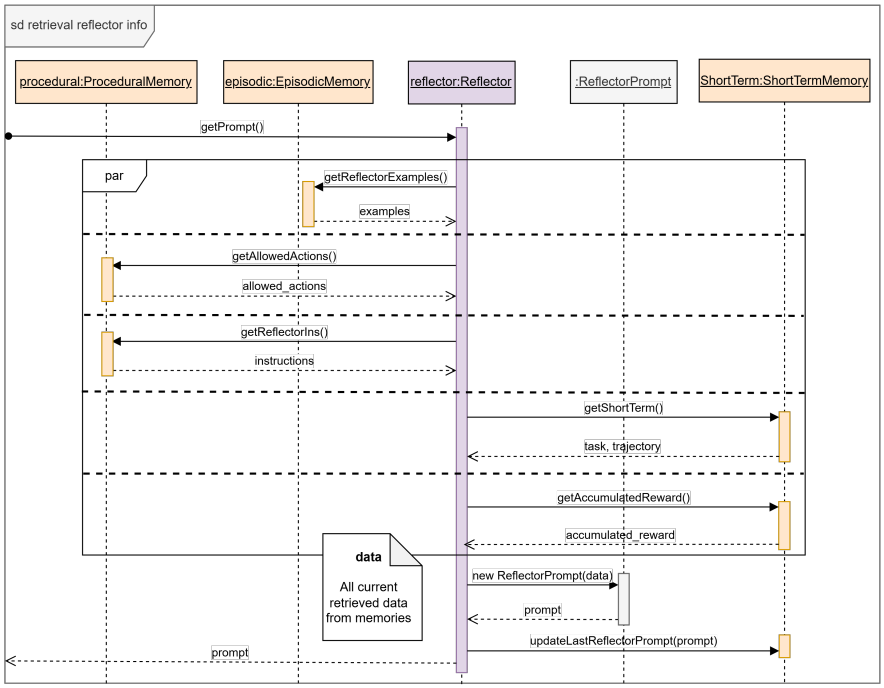

Fig. 18: UML sequence diagram of the information retrieval process for reflection in the `Retroformer` agent. In this flow, `Reflector` uses internal `retrieval` actions, represented by the *get* methods of the `episodic` and `procedural` memories, from which examples for the LLM and the allowed actions along with instructions to guide the LLM's behavior are obtained. These, together with the current task and trajectory retrieved from the `short-term memory`, are used to generate the prompt. This prompt, before being returned, is stored in the `short-term memory`.

Training Process. `Retroformer` employs an RLHF-based training method [1], where it trains a neural network model using supervised learning to evaluate reflections, along with the supervised learning-based training of the LLM contained in `Reflector`. Figure 19 illustrates the general training process of a `Retroformer` agent. This process consists of the three RLHF steps, which are abstracted into the methods *collectSFTData(train_data)*[4], *trainRewardModel()*, and *trainReflectorModel()* of the `Retroformer` agent. The three sequences are referenced in the diagram and correspond to Figures 20, 23, and 24, respectively.

The first step of the RLHF-based training focuses on data collection. Figure 20 shows how, after receiving a dictionary with environments and their respective tasks, the `Retroformer` agent iterates over each environment, updating the environment instance stored in the `Executor` component. Then, memories are initialized based on this new environment (the same process as in `Reflexion`, see Figure 3). Each task associated with the environment is iterated a total of three times. For each task, the process is quite similar to the work cycle of a `Retroformer` agent (see

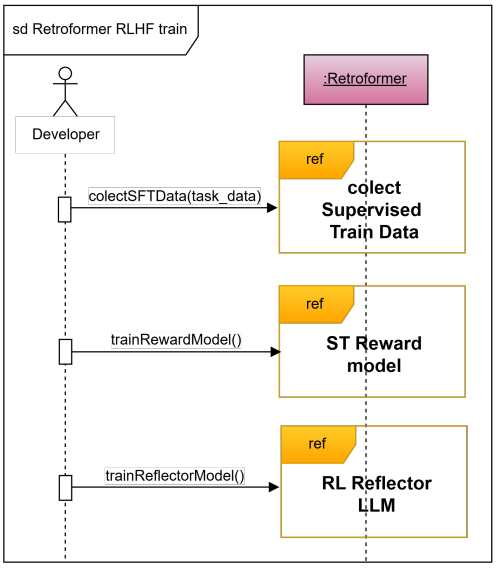

Fig. 19: Main UML sequence diagram of the RLHF-based training process of a `Retroformer` agent.

[4] The information contained in the *task_data* variable is based on a dictionary where the key represents various environments, and the value associated with each key is a set of tasks. A detailed view of the types can be found in the component description in the class diagram in Figure 12

Figure 14), where the task is solved, the response is evaluated, and the information is stored in the `short-term memory`. The difference is that if the response is incorrect, a process is initiated to generate two reflections (see Figure 21), which are then evaluated by the `Evaluator` and stored in the *replay buffer* (see Figure 22) of the `episodic memory`. Finally, the `short-term memory` is reset, and the next task is processed.

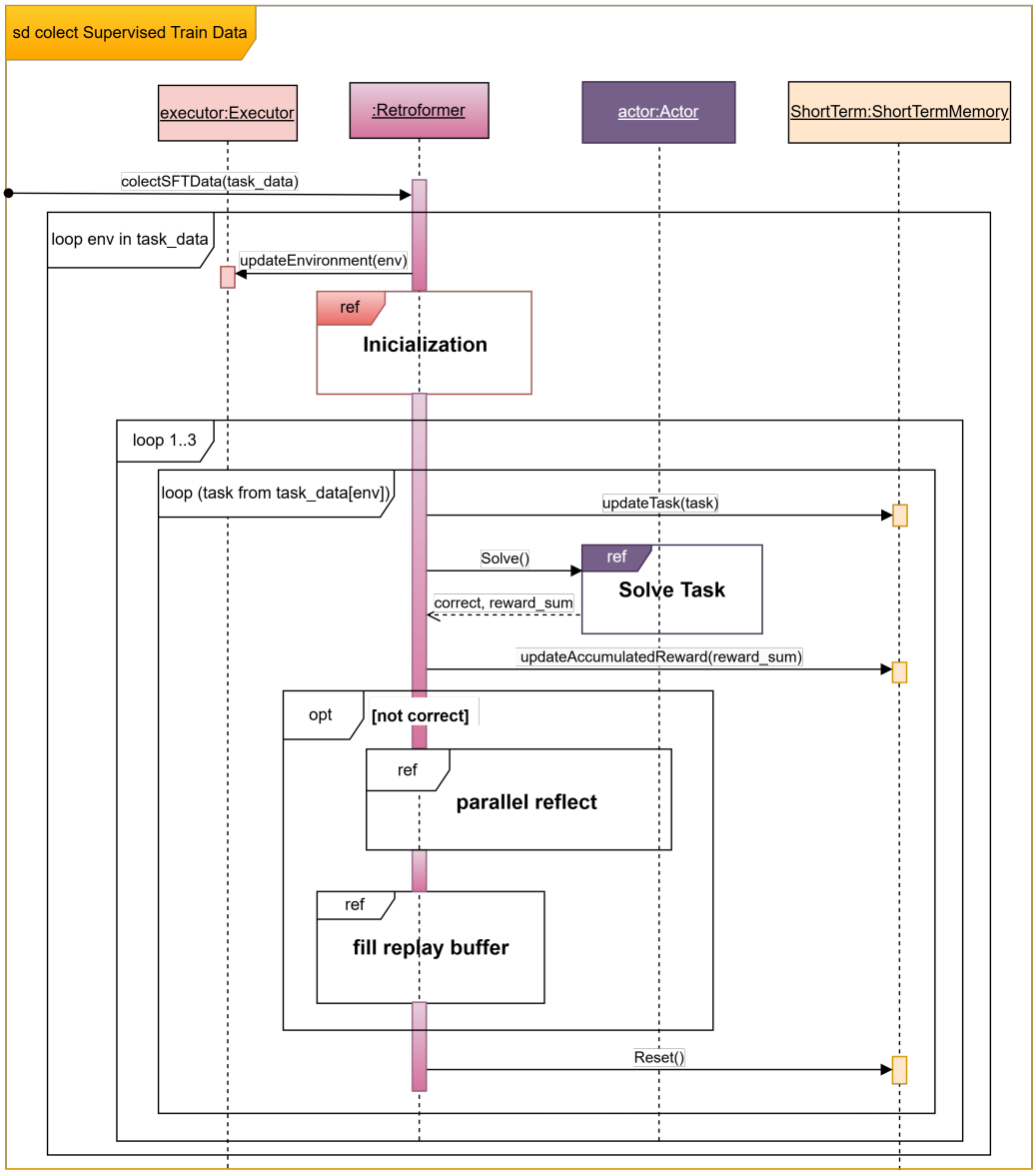

Fig. 20: UML sequence diagram of Step 1: Data collection for a `Retroformer` agent.

The process depicted in Figure 21 is another proposal from [4]. To avoid human intervention in labeling good and bad reflections, the authors of `Retroformer` propose using the difference between the accumulated rewards of two successive episodes as a signal to classify reflections into good and bad categories.[5]

[5] This is because the `Planner` LLM is considered a frozen parameter model with a low temperature. In this context, temperature refers to the parameter that configures the randomness in the LLM's response generation, implying that randomness injection is minimal. Therefore, changes in accumulated rewards are mainly attributed to the `Reflector` LLM.

In this process, based on the same trajectory, two reflections are generated. Then, each reflection is iterated, and the task is attempted again, obtaining accumulated rewards for each episode. Finally, for each of the two reflections, the prompt, reflection, and new accumulated rewards are stored in the `short-term memory` for further classification.

The final part of Step 1: Data collection is illustrated in Figure 22, which shows how the generated reflection information is stored in the *replay buffer* of the `episodic memory`. The addition of this data is based on the difference in accumulated rewards between the two consecutive episodes obtained for each reflection. Comparing this score determines which reflection is correct and which is incorrect. An undefined case by [4] is highlighted, where both scores obtained for the reflections are equal. In such a case, both reflections are classified as correct; however, it is noted that this remains an open issue.

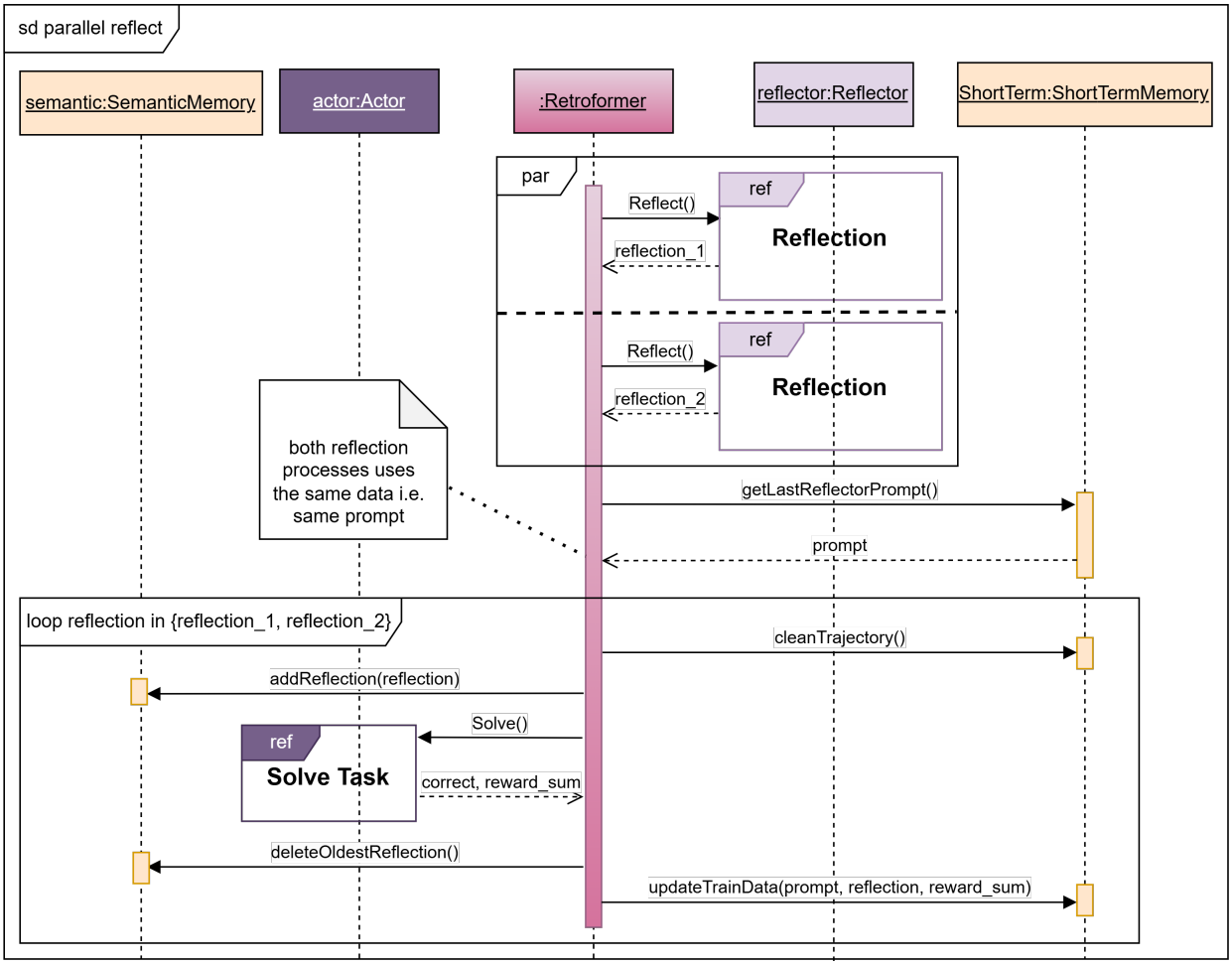

Fig. 21: UML sequence diagram of the flow that creates training data based on two reflections of a `Retroformer` agent.

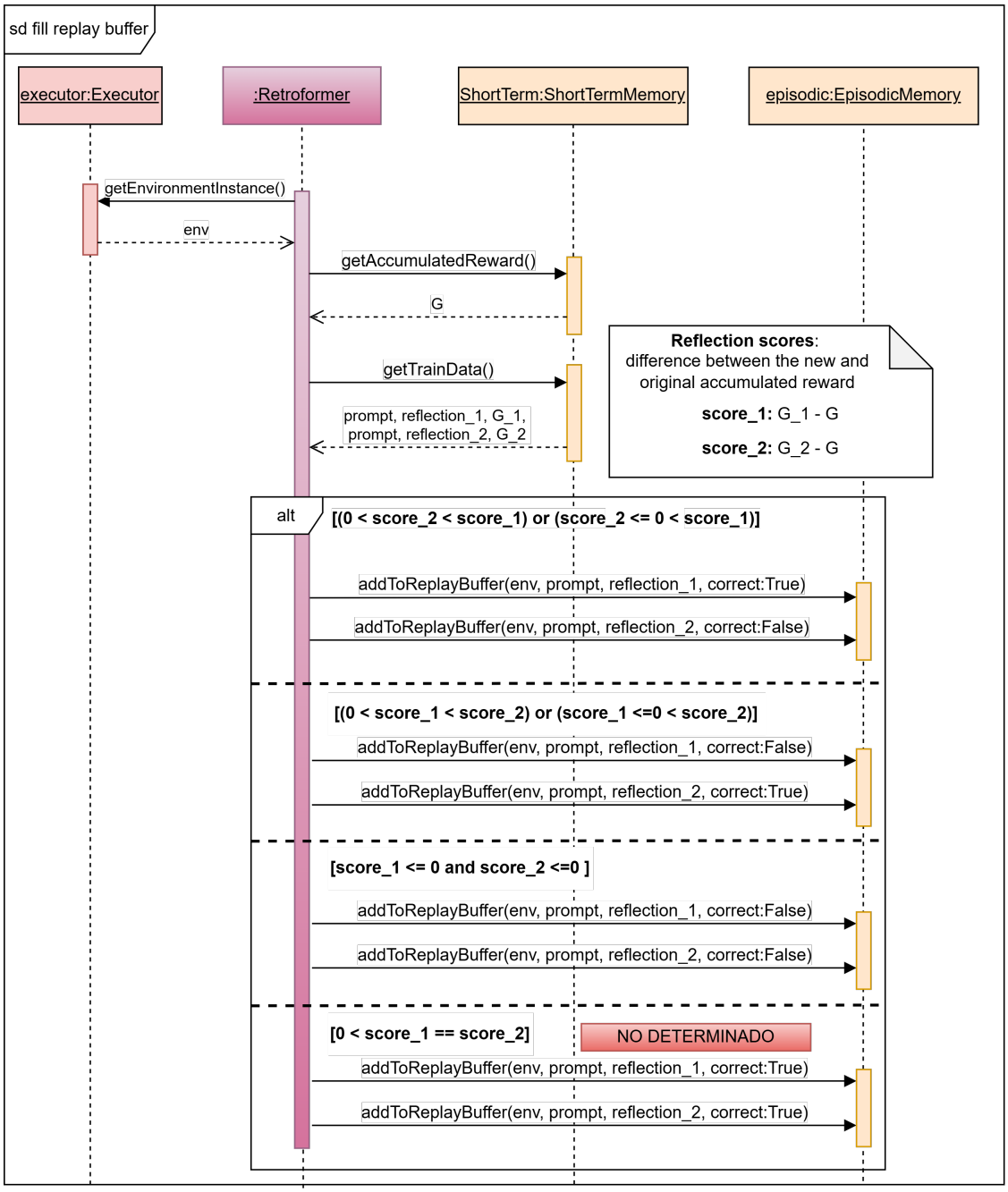

Fig. 22: UML sequence diagram showing the process of filling the *replay buffer* of the `episodic memory` in a `Retroformer` agent.

Step 2 of the training process for a `Retroformer` agent, illustrated in Figure 23, focuses on training a reward model using information from the *replay buffer* of the `episodic memory`. In this process, `Retroformer` utilizes the `procedural memory` (which implicitly stores the information contained in the parameters of both the reward model and the LLMs) to update the reward model's policy. The reward model is trained using supervised learning based on the data stored in the *replay buffer*.

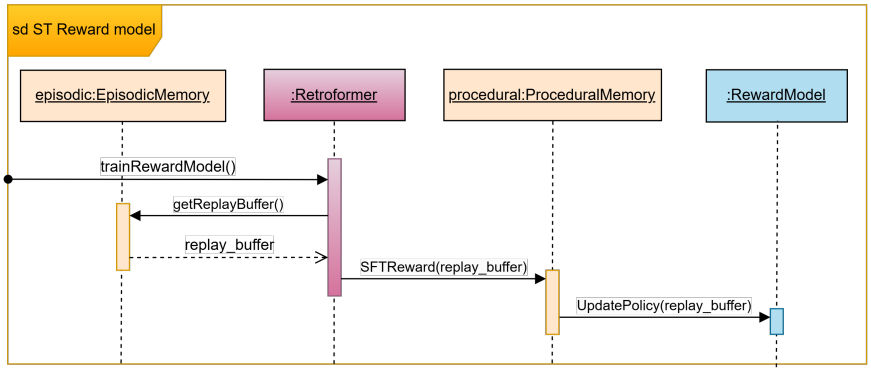

Fig. 23: UML sequence diagram of Step 2 in RLHF-based training of a neural network reward model for a `Retroformer` agent.

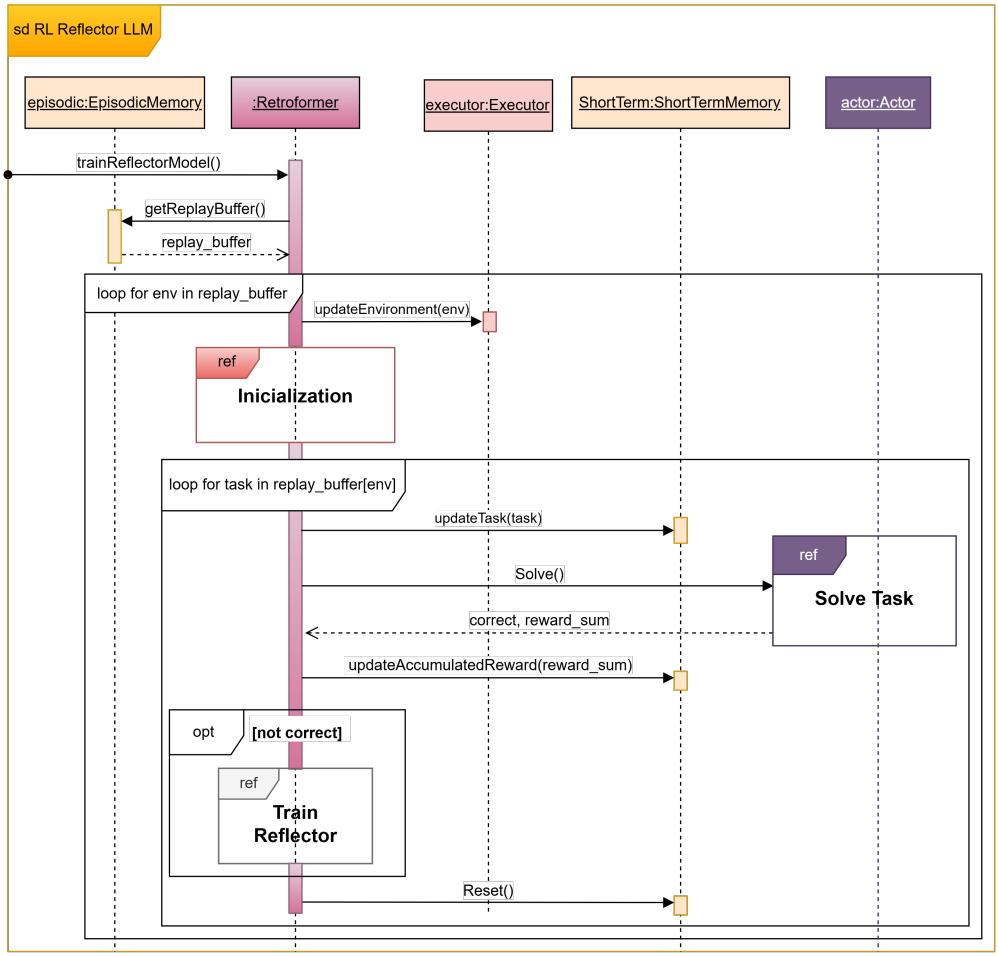

Fig. 24: UML sequence diagram of Step 3 in RLHF-based training of a `Retroformer` agent, where the LLM of the `Reflector` is trained using reinforcement learning.

Finally, Step 3 of the RLHF training process is depicted in Figure 24. In this step, the LLM of the `Reflector` is trained using reinforcement learning, leveraging the reward model trained in the previous step. The process iterates over each environment and its respective tasks stored in the *replay buffer* of the `episodic memory`.

Similar to Step 1 of data collection, the environment is initialized, and the task is solved using the `Actor`. The difference is that now, when a response fails, the process shown in Figure 25 is triggered, where a reflection is generated, evaluated, and the LLM of the `Reflector` is fine-tuned based on this reflection using the PPO algorithm [2].

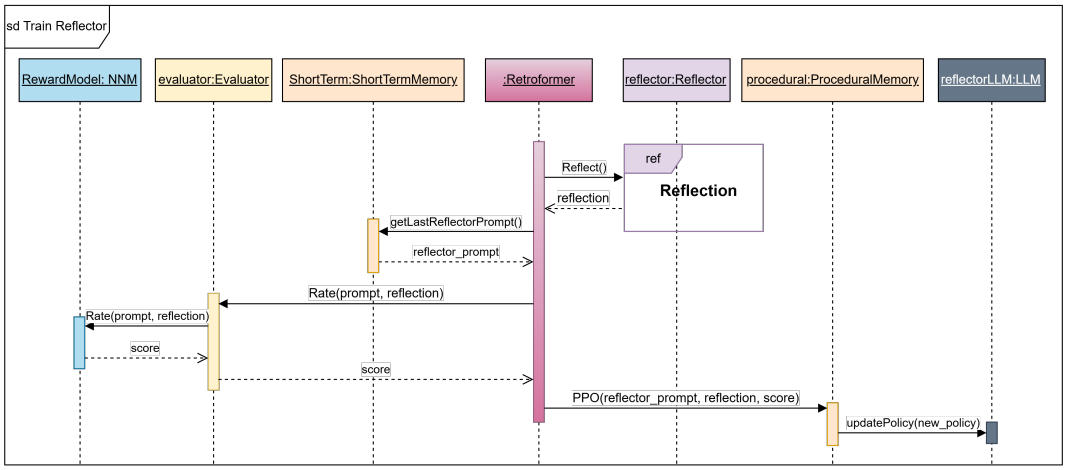

Fig. 25: UML sequence diagram where, given a reflection, it is evaluated using the reward model of `Evaluator`, and this score is then used to update the policy of the LLM in the `Reflector` component using the PPO algorithm.

(2) Formal Specification Level In the `Retroformer` architecture, we have identified seven behaviors that, to avoid ambiguities, need to be formally specified. These behaviors are:

1. The *Reset* method of the `short-term memory`, which must clear its attributes in a specific format.
2. The *isDone(Action)* method of `Evaluator`, which must return true if the received action contains the string "[answer]".
3. The `semantic memory` can store at most 3 reflections.
4. The respective precondition of the *addReflection* method in `semantic memory`, which allows adding a new reflection only if there are fewer than 3 reflections stored.
5. The trajectory invariant of the `short-term memory`, which states that stored objects must be added in the sequence Action-Observation-Reward.
6. The pre and postconditions that the *addToTrajectory* method of `short-term memory` must follow.
7. In the training process of `Retroformer`, 2 reflections are used to generate training data. We decided to explicitly define this invariant, where the information temporarily stored in the *train_data* attribute of the *short-term memory* consists of 2 reflections, one correct and one incorrect.

The behaviors 1, 2, 3, and 4, specified using OCL, are described similarly to their counterparts in `Reflexion`, which are presented in equations 1, 2, 3, and 4, respectively.

The specifications of behaviors 5, 6, and 7 are presented below in equations 7, 8, and 9, respectively.

```
context ShortTermMemory inv :
  self.trajectory → notEmpty()
  or                                                                    (7)
    let size: Integer = self.trajectory → size() in
    Sequence{1..size} → forAll(i :Integer |
      if i.mod(3) = 1 then
        self.trajectory → at(i).oclIsTypeOf(Action)
      else if i.mod(3) = 2 then
        self.trajectory → at(i).oclIsTypeOf(Observation)
      else
        self.trajectory → at(i).oclIsTypeOf(Reward)
      endif
    )
```

```
context ShortTermMemory :: addToTrajectory(newItem : TrajectoryItem): OclVoid
pre :
  if self.trajectory → isEmpty() then
    newItem.oclIsTypeOf(Action)
  else let lastElement = self.trajectory → last() in
    if lastElement.oclIsTypeOf(Action) then
      newItem.oclIsTypeOf(Observation)                                  (8)
    else if lastElement.oclIsTypeOf(Observation) then
      newItem.oclIsTypeOf(Reward)
    else if lastElement.oclIsTypeOf(Reward) then
      newItem.oclIsTypeOf(Action)
    endif
post :
  self.trajectory → size() = self.trajectory@pre → size() + 1
post :
  self.trajectory → last() = newItem
```

context `ShortTermMemory::Reset(): OclVoid`
post:
self.task = “”
post:
self.answer = “”
post:
self.trajectory → isEmpty()
post:
self.accumulated_rewards = 0
post:
self.train_data → isEmpty()
post:
self.best_reflect → isEmpty()
post:
self.last_reflector_prompt → isEmpty()

(9)

context `ShortTermMemory` **inv** :
self.train_data → size() <= 2

(10)