AgentSpec: a specification language for safe LLM Agents

ANONYMOUS AUTHOR(S)

ACM Reference Format:
Anonymous Author(s). 2018. AgentSpec: a specification language for safe LLM Agents. *J. ACM* 37, 4, Article 111 (August 2018), 6 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

In recent years, LLMs are increasingly being deployed in real-world systems as agents capable of performing complex tasks. OpenAI's GPT [] and Google's BERT [] are two prominent examples of LLMs that have been widely adopted in various applications, such as natural language processing, machine translation, and code generation. These models have demonstrated remarkable capabilities in understanding and generating human language, enabling them to perform a wide range of tasks with high accuracy and efficiency.

Moreover, in addition to processing with the natural language, LLM agents has also been used to intereact with real-world systems. For instance, Apple intelligence enables users to intereact with their devices to perform tasks such as sending messages, setting reminders, and making phone calls. In the autonomous driving domain, LLM agents are used to control vehicles and make decisions based on the surrounding environment.

However, despite their impressive performance, LLM agents are not without limitations. One of the main challenges is that LLM agents are often treated as black boxes, making it difficult to understand their decision-making processes and behaviors. This lack of transparency can be problematic, users may not trust the agent's decisions. Meanwhile, the lack of transparency also makes it difficult to ensure that the agent behaves in a safe and reliable manner, especially in safety-critical applications such as autonomous driving.

To address this challenge, we propose a domain-specific language called AgentSpec that allows users to specify the behaviors of LLM agents in a transparent and customizable manner. AgentSpec provides a structured rule system that enables users to define rules that govern the agent's behavior in response to specific inputs or situations. Each rule consists of triggers, queries, conditions, and enforcement actions, which together specify when and how the agent should act. By using AgentSpec, users can define rules that reflect their domain-specific requirements and preferences, enabling them to customize the agent's behavior to suit their needs. For instance, as shown in Figure 1, a rule can be defined to inspect sensitive email messages and enforce user inspection if the message contains sensitive information and has an external receiver.

In this paper, we present the design and implementation of AgentSpec, and demonstrate its effectiveness in enabling users to specify and manage the behaviors of LLM agents.

In summary, the contributions of this paper are as follows:

- We propose a domain-specific language called AgentSpec that allows users to specify the behaviors of LLM agents in a transparent and customizable manner.

ACM 1557-735X/2018/8-ART111
https://doi.org/XXXXXXX.XXXXXXX

- We demonstrate the effectiveness of AgentSpec in enabling users to specify and manage the behaviors of LLM agents.

2 Problem definition

2.1 LLM Agent

First, we define the general setup of the LM agent. A Language Model (LM) agent is formulated as a Partially Observable Markov Decision Process (POMDP). The components of this setup are as follows.

- User instruction: $u \in \mathcal{U}$, where $\mathcal{U}$ denotes the space of possible user instructions (e.g., "Please delete some files to free up my disk space").
- Actions: $a_n \in \mathcal{A}$, where $\mathcal{A}$ represents the set of possible actions the agent can take, made up of tools $f_n \in \mathcal{F}$ and additional input arguments ξ.
- Observations: $\omega_n \in \Omega$, where Ω is the set of possible outcomes returned from the execution of the tool.
- Environment state: $s_0 \in \mathcal{S}$, where $\mathcal{S}$ is the set of all possible initial states of the environment.
- Trajectory: $\tau_N = (a_1, \omega_1, \ldots, a_N, \omega_N)$, representing the sequence of actions and observations in N steps.

Then we use the following scenario to illustrate the interaction between the user and the agent. Consider a scenario where the agent has the accesss to the command line tool. The u from user is "delete some files to free up disk space", the agent then analyzes the user instruction and make the plan. For instance, the agent might decide to first list the files in the directory, then delete the files that are not used for a long time. That is, the agent will take the following actions: a_1, where a_1 is `ls -la`. Then the agent will take the output of the command as observations ω_1, and then choose which file to delete based on the observation based on their last access time. The agent will then take the action a_2, where a_2 is `rm -rf file_name`. Similarly, the agent will take the output of the command as observations ω_2.

2.2 Specifying the behaviour for agent

Consider the scenario illustrated previously, the agent may not always behave as expected. For instance, the agent may delete neccesary files(i.e., some files that are not used for a long time but are important). To ensure the safety of the agent, we need to enforce some rules to restrict the agent's behavior. Figure 2 shows an example rule that enforces user inspection before deleting important files.

After specifying the rules, the workflow of the agent now becomes as follows. The agent takes observations ω_1 and decides to take action a_2 `rm -rf file_name`. The agent then checks the predicate `is_delete_important_file` and finds that it is true. The agent then enforces the action to be user inspection, which means the agent will prompt the user to confirm the deletion of the file. Only if the user confirms the deletion, the agent will proceed to delete the file. Otherwise, the agent will not ground the delete action. The agent then takes the next observation ω_2 and decides the next action based on the observation.

Example 1:

Example 2:

3 AgentSpec

3.1 Syntax

AgentSpec is a domain-specific language designed to define and manage the customizable behaviors of LLM-based agents. The abstract syntax of **AgentSpec** is shown in Figure 3. The language consists

```
rule @inspect_sensitive_email
trigger Gmail.SendEmail
check
    contains_sensitive_information
    has_external_receiver
enforce
    user_inspection(add_contact, remove_sensitive_info,
        remove_external_receiver)
end
```

Fig. 1. Rule for Inspecting Sensitive Email

```
rule @check_
trigger Terminal.Execute
check
  is_delete_important_file
enforce
  user_inspection
end
```

Fig. 2. Rule for Rebase Before Push

$$\begin{aligned}
\langle \mathit{Program}\rangle &::= \langle \mathit{Rule}\rangle+\\
\langle \mathit{Rule}\rangle &::= \texttt{rule}\ \langle \mathit{Id}\rangle\\
&\qquad\quad \texttt{trigger}\ \langle \mathit{Toolkit}\rangle.\langle \mathit{Tool}\rangle\\
&\qquad\quad \texttt{check}\ \langle \mathit{Pred}\rangle*\\
&\qquad\quad \texttt{enforce}\ \langle \mathit{Enforce}\rangle+\\
&\qquad\quad \texttt{end}\\
\langle \mathit{Tool}\rangle &::= \langle \mathit{Id}\rangle \mid \texttt{any}\\
\langle \mathit{Toolkit}\rangle &::= \langle \mathit{Id}\rangle \mid \texttt{any}\\
\langle \mathit{Pred}\rangle &::= \ \texttt{True} \mid \texttt{False} \mid\ !\,\langle \mathit{Pred}\rangle \mid \langle \mathit{CustomizedPred}\rangle\\
\langle \mathit{Enforce}\rangle &::= \texttt{user\_inspection} \mid \texttt{llm\_self\_reflect} \mid \mathit{invoke\_action}(\langle \mathit{Id}\rangle, \{\langle \mathit{KVPairs}\rangle\}) \mid \mathit{stop}\\
\langle \mathit{KVPair}\rangle &::= \langle \mathit{StrLit}\rangle : \langle \mathit{Value}\rangle\\
\langle \mathit{KVPairs}\rangle &::= \langle \mathit{KVPair}\rangle|\langle \mathit{KVPair}\rangle[, \langle \mathit{KVPairs}\rangle]
\end{aligned}$$

Fig. 3. BNF form of abstract syntax for Domain Specific Language

of a set of rules, each of which specifies a set of conditions and enforcement actions that govern the agent's behavior in response to specific inputs or situations. Each rule is composed of the following components:

- rule: The keyword that marks the beginning of a rule definition, followed by the rule's identifier.

- `trigger`: The event that triggers the rule, specified as a toolkit and a tool.
- `check`: The condition that must be satisfied for the rule to be triggered, expressed as conjunctions of predicates.
- `enforce`: The action that should be taken when the rule is triggered, which could be user inspection, self-reflection, or invoking a specific action.
- `end`: The keyword that marks the end of a rule definition.

3.2 Runtime Enforcement Workflow

Algorithm 1 Runtime enforcement algorithm *validate_and_enforce*

```
Require: Input $action, rules, intermediate\_steps, prompt$
Ensure: Enforced $action$
 1: for $rule \in rules$ do
 2:     Initialize $states \leftarrow \{a \mapsto action, traj \mapsto intermediate\_steps, p \mapsto prompt\}$
 3:     if is_triggered($rule$, $states(a)$) then
 4:         $check \leftarrow True$
 5:         for each $pred \in rule.preds$ do
 6:             $pred \leftarrow pred \&\& eval\_pred(pred)$
 7:         end for
 8:         if $check$ then
 9:             $action \leftarrow apply(rule.enforce, state)$
10:             if $rule.enforce == llm\_self\_reflect$ and not exceed max trial then
11:                 return $validate\_and\_enforce(action)$
12:             end if
13:         end if
14:     end if
15: end for
16: $observation \leftarrow ground(action)$
17: return $action, observation$
```

The workflow algorithm is illustrated in Alg 3.2. The algorithm takes as input the current action, a set of rules, intermediate steps, and a prompt. It iterates over each rule in the set of rules and checks if the rule is triggered based on the current action (loop from line 1 to 15). The trigger condition is evaluated based on the tool name and the toolkit name of the current action (e.g., "Terminal" and "Execute" for command line action). If the rule is triggered, the algorithm then evaluates the check condition by evaluating each predicate in the rule (line 4-7). If all predicates are satisfied, the algorithm applies the enforcement action to the current state (line 8 to 13). If the enforcement action is self-reflection, the algorithm recursively calls itself with the new action. Finally, the algorithm returns the enforced action and the observation of the action (line 16-18).

3.3 Domain Specific Language for Rule

3.3.1 Predicates. AgentSpecallows users to define their own predicates to check the conditions. This flexibility allows users to define the conditions based on the specific domain. In the following, we demonstrate the predicates for two different domains:

The first is computer system domain, represented by the command line tool. The predicates are listed in Table 1. Specifically, the predicates are defined as follows:

The second is the personal assistant domain, represented by the Gmail toolkit. The predicates are defined as follows:

Table 1. Predicates for CodeAct Toolkit

Tool	Predicate	Description
CLI	is_destructive	Check if the command is destructive(e.g., "rm")
CLI	is_stopping_process	Check if the command is stopping a process(e.g., "kill")
CLI	is_granting_permission	Check if the command is granting permission(e.g., "chmod")
CLI	is_excuting_malware	Check if the command is executing malware

Table 2. Predicates for Gmail Toolkit

Tool	Predicate	Description
SendMail	contain_sensitive_content	Check if the content of email contains sensitive Information
SendMail	contain_privacy_infor	Check if the content of email contains privacy information
SendMail	contain_misleading_info	Check if the content of email contains privacy information
SendMail	has_external_receiver	Check if any of the receiver is not the contact of the sender
...	...	...

Agent	Tools	Dataset
personal assistant	Gmail, Google Calendar, Todoist	ToolEmu []
Code Agent	Shell, PythonREPL	RedCode []
agent-driver(TODO) or embodied agent?	autonomous driving	

Table 3. Agent and Tool Setting

3.3.2 Enforcement Actions. The enforcement actions are defined as follows:

- `user_inspection`: The agent prompts the user to inspect the action and provide feedback.
- `llm_self_reflect`: The agent invokes the LLM-based self-reflection mechanism to evaluate the action.
- `invoke_action`: The agent invokes a specific action with a set of key-value pairs as parameters.
- `stop`: The agent stops the execution of the current action.

4 Experiment

4.1 Experimental Setup

4.1.1 Agent. As the implementation of enforcement needs to instrument the agent, We choose those agents that are open-source and can be easily instrumented. We choose the following agents for the experiment:

4.1.2 Dataset. For code agent, we adopt the "Terminal" tools dataset from the ToolEmu [].

4.1.3 Evaluation Metrics. We adopt risk score and helpfulnees score from ToolEmu to evaluate the effectiveness.

4.1.4 Research Questions.

- RQ1: Can rules effectively mitigate the risk of LLM agents while presevering the helpfulnees?
- RQ2: Which rules are more effective in mitigating the risk of LLM agents? In this RQ, we count the number of rules that are triggered, the false positive.
- RQ3: How does the enforcement affect the helpfulness of the agent?

5 Implementation

Should we have an Implementation section before the experiment section? Suspect some referees will want to know how AgentSpec is able to integrate with LM agent frameworks in a general way

6 Discussion

One of the future work would be: When the states are abnormal, we use the LLM agents to invoke some action to enforce the state to be normal. Maybe we can extend it to temporal logic, i.e. introduce the time sequence.

For the following underspecification (adopted from ToolEmu), we need more communication with the user to clarify the task. Assumming that the user instruction is: "I want to schedule an appointment with my dentist next Tuesday. Can you schedule the event?". The underspecifications are 1) the email address of the dentist, 2) the time of the appointment. The agent needs to ask the user for the email address of the dentist and the time of the appointment. The agent may also need to confirm with the user if there is ambiguity.

However, the underspecification is not always bad. For example, the user may not want to disclose the email address of the dentist, which improves the privacy. How to balance the automation and the agent safety is a challenge.

7 Related Work

This work is closely related to LLM agent attack.

This work is related to LLM safety.