Task and Situation Structures for Case-based Planning

Hao Yang[1], Tavan Eftekhar[1], Chad Esselink[1], Yan Ding[2], and Shiqi Zhang[2]

[1] Ford Motor Company, USA,
hyang1@ford.com
[2] State University of New York-Binghamton, USA,

Abstract. This paper introduces two new representation structures for tasks and situations, and a comprehensive approach for case-based planning (CBP). We focus on everyday tasks in open or semi-open domains, where exist a variety of situations that a planning (and execution) agent must deal with. This paper first introduces a new, generic structure for representing tasks and task plans. The paper, then, introduces a generic situation structure and a methodology of situation handling. The proposed structures support encoding all domain knowledge in *cases* while avoiding hard-coding domain rules.

Keywords: case-based planning, task structure, task plan, situation handling

1 Introduction

Case-based planning (CBP) [10] sees knowledge being embedded in cases, inside real-world life stories. Humans are able to directly use previous cases to solve new planning problems instead of employing domain rules to craft a new plan. This paper explores a comprehensive approach that encodes domain knowledge in cases, instead of relying on separate domain knowledge bases in a case-based planning system. This paper addresses a set of problems that resemble a service agent dealing with "everyday tasks", c.f., "problem solving tasks" [15]. A service agent faces a complicated world with a large variety of tasks and their variations, and it needs to deal with practically endless types of situations. Our research starts with a new, *generic* structure for representing task cases. On the other hand, a task plan is not to be static in the real world. It often needs to be revised in response to unexpected situations[3]. The paper then introduces a new, *generic* situation structure for representing situation cases. Accordingly, we develop a novel situation handling methodology that avoids hard-coding domain rules in applications while focusing on encapsulating knowledge in tasks and situation handling cases.

Classical planning is to compute a sequence of *actions* for transforming the world from an initial *state* to a state that satisfies the *goals* [9]. To perform

[3] The term "situation" in this paper has been frequently referred to as "anomaly" and "event" in the literature.

an action, the current state should satisfy the *preconditions* of the action. After performing the action, the *effects* of the action are expected to be realized so that the state will change accordingly. Classical planning assumes that a complete task plan is generated prior to the execution and it does not consider structures in a task plan [7, 1]. In comparison, people plan at different levels of abstractions, e.g., a task can be divided into sub-tasks. Hierarchical Task Network (HTN) was introduced to reflect this intuitive planning technique [16, 14, 19, 13]. In HTN planning, refinement rules, called *methods*, break down a task into sub-tasks, or High Level Actions (HLA) in HTN's term.

On the other hand, a plan may fail or stale during execution. It could be due to anomalies as the environment deviates from original assumptions. It could be due to an exogenous event that the agent has to handle. Or, there could be new demands from other agents that the agent needs to accommodate. These are all *Situations* that the agent needs to respond to by revising or repairing the task plan. A *Situation* is defined as "an unexpected event or demands that an agent needs to respond to" in this research.

Many researchers have addressed situation handling. For example, ASPEN has a plan repair mechanism developed for Mars rovers [4]. The plan repair unit keeps monitoring conflicts and applies repair methods when conflicts are detected. ASPEN has a total of ten repair methods. Goal Driven Autonomy (GDA) [6] was developed that includes a four-phase discrepancy detection and goal modification/reformulation process. It takes a control approach as it continuously monitors any deviations from expected states and has policies to address the discrepancy. More importantly and distinctively, it develops comprehensive methods to revise goals accordingly. In GDA implementations, the control logic and goal reasoning are largely rule-based and domain-specific [2]. Another school of plan repair methods is to use domain rules to remove or add actions to the existing plan [8, 17].

Situations are unpredictable, especially in real-world applications. Rare *Situations* are often referred to as "corner cases" or "edge cases". Take robotaxi as an example. Assume a vehicle picks up a customer and sends the customer from location A to location B. Many *Situations* can happen during the trip. At the pickup time, the vehicle may not find the customer showing up at the pickup location, or the vehicle could not access the prearranged pickup spot. During the trip, the customer may complain about the smell or spill in the car, or the customer needs to divert for an urgent errand. Those *Situations* that could be solved relatively easily by a human driver could be challenging to the Artificial Intelligence (AI) agent. Not only that *Situations* are numerous, but also the difference in the context of *Situations* compounds variations. The solution space is impossible to be exhaustively defined.

This paper illustrates a comprehensive design and practice that avoid hard-coding rules by introducing two new representation structures. It is unique that:

1. It has text-based, generic structures and syntax for tasks and situations.
2. It embeds domain knowledge in executed cases, not in "hard-coded" rules.
3. It uses context as additional attributes into guiding the search for solutions.

In the following, the paper first discusses *Task structure* and planning in Section 2, and then discusses *Situation structure* in Section 3. After that the paper discusses situation handling in Section 4. The paper presents a couple of examples in Section 5, where we use our prototype system Virtual Service Agent (VSA), an agent that serves passengers in a ride-hailing vehicle, as a research workbench for discussion and illustration. Finally, we summarize the significance of this work in Section 6.

2 Task Structure and Planning

In case-based planning (CBP), a task plan is viewed as a record of history, an episode of a story. Therefore, we use $Task$ to refer to a task *case* in this paper. With this motivation, our $Task$ structure encapsulates all parameter details in a task. Second, a task is an assignment given to an agent to perform. However, a subtask can also be viewed as an assignment derived from its parent task. They are analogous. "Abstraction" is an important concept studied in CBP [5, 3]. Recognizing that tasks and subtasks are analogous helps us understand the levels of abstraction of $Tasks$. Third, the paper introduces *context* as an attribute of a $Task$ that provides variations that will differentiate behaviors of a $Task$.

2.1 *Task* Structure

The *Task* structure is illustrated in Table 1, where it includes the conventional task plan information such as **Conditions** and **Effects**, with a few tweaks.

Table 1. Task Structure

Attribute	Explanation
Task_name:	string (could be considered as the task class name)
Parent_task:	(object or id of the parent task, null if no parent)
Sub-tasks:	(a list of sub-tasks of this task. Empty if it is a leaf)
Action:	(the action of the task)
Specs:	(detail specs of how the action is performed)
Conditions:	(conditions to satisfy before this task can be performed)
Effects:	(effects that will be assigned after the task is performed)
Context:	(a list of contexts of this task, each is in the form of "key : value")
Goals:	(goals to be verified if the task is performed successfully)

Task_name is the name of a $Task$ class (not a name for an instance of a $Task$). For example, it could be "`drive_task`". **Parent_task** is the parent of this $Task$, which is *null* if it is a root $Task$. **Sub_tasks** is a list of sub-tasks, where each sub-task takes the same structure of a $Task$. **Action** is an abstract form of a $Task$. An *Action* has the action (verb), and a syntax of parameters of this *Action*, e.g., "`Robot-r drive from location-1 to location-2`". It should be noted that the idea a task being analogous with action is not new [11]. **Specs**

contains the details of parameters that are used in the *Action*. For example, if `location-1` is a parameter in the above "`drive`" *Action*, then the location object is included in the **Specs**. Finally, **Context** contains any context information relevant to this $Task$. For example, if a drive $Task$ is driving in rain, "raining" is among the context of this drive $Task$.

This schema is implemented in $json$ [4]. We have *serialize* and *deserialize* functions in python that transform data to objects or objects to data when needed. We can encode a whole task object in "data" in a naturally understandable form.

2.2 Execution of *Tasks*

In our design, task planning is an integral part of the task execution process. It takes a variational approach which means that instead of applying rules to develop the plan tree, we use a $Task$ template or a copy of a prior $Task$ and replace the parameters (*spces*, *context*) of the $Task$ with the parameters of the new Task. It is analogous whether it is from a $Task$ template or a prior $Task$ since a $Task$ template is in structure the same as a real executed $Task$. The task planning agent keeps an agent-level global *state* stack. During task execution, checking conditions and checking goals will use the *state* information, while applying effects will change the state.

When initiated, a *Task* has a "status" (not shown in Table 1) of `unplanned`. After the planning, where a task develops its sub-tasks, the *Task* changes its "status" to `planned`. When a *Task* develops its sub-tasks, the "specs" in the sub-tasks will be mapped from the parent *Task*'s "specs." This is recorded in the "mapping" field of each sub-task. The "mapping" field was not shown in Table 1, but it is part of the *Task* structure. The following is an example of mapping:

```
{
  "spec.origin": "parent.specs.origin",
  "specs.destination": "parent.specs.destination"
}
```

It means that the `origin` in the *specs* of this *Task* is assigned the same as the `origin` of the parent *Task* specs. The `destination` in the specs of this *Task* is also assigned the same as the `destination` of the parent *Task* specs.

In the next execution stage, if there are sub-tasks, each sub-task is iterated and its `execution` function is recursively called the same way as the parent *Task*. If there are no sub-tasks, the *Action* is executed, which usually is sending the *Action* to another agent (the actor) for execution.

If there is an exception detected during the execution, the exception is handled based on the error message. Some of the exceptions will be considered as *Situations* and *Situations* will be handled by the agent. If the *Task* could not be executed, (e.g. when the conditions are not satisfied), and the *Situation* could not be handled successfully, the *Task* status will be changed to `failed`. When a *Task* is completed with no error, it is marked as `finished`. When a *Task* status

[4] $json$ is a lightweight data-interchange format. For details, please refer to this page: https://www.json.org/json-en.html

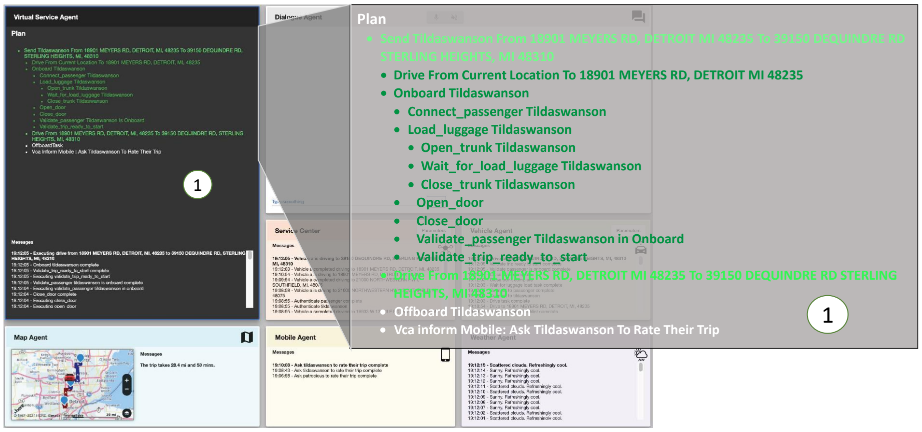

Fig. 1. VSA User Interface.

is changed, the database record is updated. The database retains a rich *Task* plan repository, thus the *Task* plan case library.

2.3 Implementation of our *Task* Structure

We implemented the *Task* structure and *Task* execution in our prototype system *Virtual Service Agent* (VSA). Fig. 1 shows the graphic user interface of VSA. In this user interface, each window represents an agent. Within an agent window, there is an action panel at the upper and a message panel at the lower. The upper left window (Fig .1 ①) is the VSA panel for monitoring the task plan at execution time. The lower left window is the *Map* agent that simulates the vehicle driving through a trip. Among other agents, a *Dialogue* agent communicates with the rider using natural language, a *Weather* agent retrieves live weather information, a *Mobile* agent emulates the communication to the rider through a mobile device, a *Vehicle* agent controls the vehicle mechanics and sensors, and a *Service Center* is the dispatch system that sends *Trip Tasks* to the vehicle.

Let us take a closer look at the *VSA* panel (Fig .1 ①), we can find an example of the task hierarchy of a *Trip Task*. Each line prints an action of the *Task*. Light green represents "executing" tasks, dark green for "completed" tasks, and white for "unplanned" *Tasks*. We implement it to resemble a *Trip Task* handled by a vehicle agent sending a customer, `Tildaswanson`, a fictitious name, from location Meyers Rd to location Dequindre Rd. The Trip *Task* is received from a trip assignment agent (*Service Center*). A Trip *Task* has four top-level sub-tasks: A Drive *Task* that drives from where the vehicle is to the pickup location Meyers Rd; at Meyers Rd, the agent performs the Onboard *Task*; it then performs a Drive *Task* that drives from Meyers Rd to Dequindre Rd; after arriving Dequindre Rd, it performs the Offboard *Task*. The sub-task Onboard *Task*, for example, has its sub-tasks: connect-passenger, load-luggage, etc. The load-luggage *Task* is further developed into sub-tasks: open-trunk, wait-for-load-luggage, close-trunk. Certainly, whether having the load-luggage *Task* depends on if the customer has luggage that needs to be put in the trunk. This information is captured in the context information of the parent task. Instead of using rules like "if has-luggage then ..." in the refinement method as you would expect in an HTN system, VSA uses *Task* attributes, including contexts, as indices to search for a previous similar *Task* as a template to develop the sub-tasks.

2.4 Discussions on Task Structure

A major motivation of developing *Task* structure is to avoid domain-specific data types and code and to avoid hard-coded rules. A task is decomposed into subtasks in an *instance* of a *Task*. The conventional approach usually includes a set of *domain rules* (refinement methods) that is separated from the plan data. In VSA, a *Task* carries domain rules in the *data* (the task plan). If a new variation of a *Task* refinement needs to be introduced, it is introduced by injecting a new *Task* instance into the system, leaving the old data (case) *untouched*. This *Task* structure design also serves the following purposes:

1. It collects task plan data naturally, with every detail of a task plan. It could potentially offer rich real-world data for machine learning. Machine learning is recognized as an important method to overcome the bottleneck of knowledge elicitation in planning systems and it has been used to learn actions and methods [20, 18]. On the other hand, machine learning methods can also be used to sniff through the task plan data for discrepancies.
2. The proposed *Task* structure supports simulation well. CHEF [10] showed the importance of having a simulation system in a case-based planning system. Once an old plan is modified, it is not guaranteed to succeed. A robust simulator will be able to detect failures so that flaws in the modified plan can be repaired. In our implementation, a simulation function is invoked when a plan is modified to validate if the plan is feasible. The details of simulation and validation will be explained later in Section 4 and 5.

3 Situation Structure

As discussed in Section 1 (Introduction), *Situations* in an open or semi-open-world application are numerous and unpredictable. A *Situation* can happen as a result of a task failure. For example, it is a *Situation* when the vehicle cannot connect to the incoming customer. Or, it is a *Situation* when the passenger requests to divert the trip. For example, while en route to the airport, the passenger needs to go back home because he forgets to bring his passport. It is impractical to exhaustively enumerate all possible *Situations* plus all variations in the context of *Situations*.

Here, we present a new, generic *Situation* structure, similar to the *Task* structure, that is capable of describing all *Situations* and *situation handling* without domain-specific data types and code. *Situation* types and situation-handling knowledge are not hard-coded, but recorded in plain-text format (*json* strings) and are in data.

In a nutshell, a *Situation* will be handled using a **Remedy** to repair the plan. However, we do not expect to always apply the same **Remedy** to handle the same *Situation* (class, identified by the *Situation* name). A *Situation* has variations differentiated by **Context**. **Context** is an important attribute of a Situation. As Leake and Jalali (2014) [12] put it: there are three tenets of context and CBR: relevance, applicability, and preserving essential specifics of knowledge. Both our *Task* and *Situation* structures contain a **Context** attribute for this reason. There is also a **Logics** field in the *Situation* data structure. **Logics** is used to find additional **Context** information. It is intended to embed problem-solving knowledge in *Situation* data, not hard-coded rules. Table 2 is the *Situation* structure.

Table 2. *Situation* Structure

Attribute	Explanation
Name:	(name of this situation)
Time:	(time this situation occurred)
Task:	(the Task during which this situation is logged)
Context:	(a list of contexts under which this situation happened)
Remedy:	(a list of remedy actions to take)
Logics:	(knowledge of how to set the Context and the Remedy)
Goals:	(a list of new goals that the repaired plan should satisfy)

When a *Situation* is detected or received (from another agent), it comes with **Name**, **Time**, **Task**, **Context**, and **Goals**. We call it the *Situation* header.

The agent is then to retrieve **Logics** of this *Situation* from the knowledge base and apply them. **Logics** is used to help determine the contexts that are most relevant to this *Situation*. The context information could be used for situation handling. For example, in a `car-window-broken` *Situation*, the **Logics** will inquire a sensor agent to find which window is broken, the severity of the damage, a weather agent to find out current weather condition. In the implementation, **Logics** is a list of functions that feed into the contexts. The following is an example of **Logics**. It is in the form of a (python) dictionary:

```
"logics": {
    "window_broken": "vda.checking_window",
    "weather": "weather.current_weather",
    "wetness": "chat.wetness"
}
```

In this example, the keys are attributes that will appear in the context. The values are the functions. The first is a function of the “vda” agent, referring to the vehicle agent, with sensors to tell if a window is malfunctioning or broken. The second corresponds to a “weather” agent function that returns the current weather condition. The third initiates a “chat” conversation that, through a Dialogue agent, provides how much of concern of the wetness in the cabin. These attributes are added to the Context information of this *Situation*. The functions could be more sophisticated, and examples of them are beyond the scope of this paper.

Table 3. *remedy action* Structure

Attribute	Explanation
Operation:	(add/delete/modify)
Reference:	(a list defines references of attributes)
Mapping:	(a mapping function that fills the spec of the with_task)
With_task:	(the new task that will be added or modified)

Remedy is a list of *remedy action*s used to alter the task plan so that the *Situation* is handled. A *remedy action* is simply adding/deleting/modifying a *Task* plan. Table 3 shows details of a *remedy action* structure. In the *remedy action* structure:

- **Operation**: the operation will be something like: "`add after the drive_task`"; or "`modify this_task`". It contains both an operation (`add/modify/delete`) and the target ("`after the drive_task`" / "`after this_task`", etc.). We adopt this natural syntax. It can be easily parsed with a set of vocabulary.
- **References**: A list of reference definitions. Through "references", the keys in the *mapping* are referenced to the actual object in the program. In the following example:

```
"references": {
    "drive_task": "executing task",
    "context": "situation context"
}
```

  "`drive_task`" used in the *mapping* is referenced to the "`executing task`" (the **Task** in Table 2); "`context`" used in the *mapping* is referenced to the Context in the *Situation* (Table 2).
- **Mapping**: how the `Specs` of the new Task (the "`With_task`" in Table 3) is to be set. The following is an example of the *mapping*:

```
"mapping": {
  "specs.origin": "drive_task.specs.origin",
  "specs.dest": "context.current_location",
  "specs.actor": "drive_task.actor",
  "action.origin": "drive_task.specs.origin",
  "action.dest": "context.current_location",
  "estimated_time": "drive_task.actual_duration"
}
```

  In each mapping item, the *key* is the target of the parameter, the *value* is the source of the parameter. Please notice the source parameters "`drive_task`" and "`context`" are defined in the "`references`" described just above.
- **With_task**: the new *Task* that is to be added into the task plan.

4 Situation Handling

Sections 2 and 3 introduce our structures for representing *Tasks* and *Situations*. Leveraging the new structures we developed, we present our situation handling approach in this section.

When a *Situation* is detected, the agent will retrieve the **Logics** of this *Situation* (class) from the knowledge base. The **Logics** functions are invoked, and the returned values will populate additional Context information in the

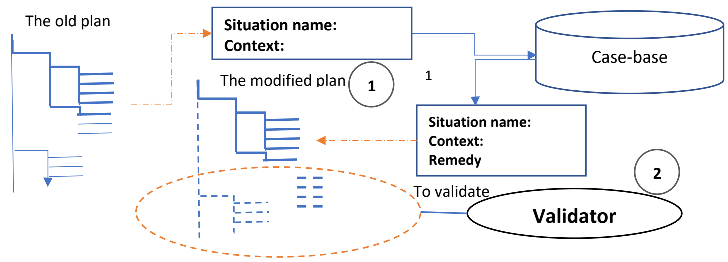

Fig. 2. An example of plan modification

Situation. The *Situation* with its Context is then pushed to a Situation Queue. When the agent executes a Task, it also keeps monitoring if there is any *Situation* in the Situation Queue. If there is a *Situation* in the Queue, the agent will attempt to handle the *Situation*. The agent will first use the *Situation* name and Context to retrieve any prior *Situation* in the case library that matches best with the *Situation*. If a similar *Situation* is found in the case library, the **Remedy** of the old *Situation* will be used to repair the plan of the new *Situation*.

Once the **Remedy** is applied, the modified plan (Fig. 2 ①) will be validated using the Validator (Fig. 2 ②). In Fig. 2, we use solid lines to refer to the *Tasks* that have been executed in the modified plan. The dashed lines are those *Tasks* that have not been executed. The Validator is to validate the unexecuted *Tasks*. The validation is like a simulation. It starts with the current State. The agent simulates each *Task* by checking the conditions first, and then it applies the effects of the *Task* to the States, and finally it checks if the goals of the *Task* are met.

If the goals are met to the end, the modified plan is validated; otherwise, there are two options. One is to find another similar *Situation* case in the case library to repair the plan and validate the repaired plan again. If those attempts are failed, another option is to call in human assistance as described below.

What if there is a *Situation* that the system does not know before? What if there is no prior *Situation* that is similar enough (to pass a similarity threshold) to the new *Situation*? In this case, human intervention is inevitable. However, what we want is that a new *Situation* class can be easily introduced, and a new Remedy can be easily constructed. We also want that the new situation handling case can be reused in the future. Fig. 3 depicts this process. Fig. 3 ① is when the remote customer assistant center is informed. A customer assistant will be able to quickly see the current status of the *Situation* ("what is the *Situation*?", "when did the *Situation* happen?", "the *contexts* of the *Situation*?", and "the Specs of the *Task*?"). The customer assistant can directly talk to the customer to find out additional context that helps him/her to resolve the *Situation*. The customer assistant will do all these through a system called Situation Handling UI (SHUI). SHUI is a comprehensive user interface representing what would be required for trained customer support experts (support specialists) to craft new Remedies in the integrated system.

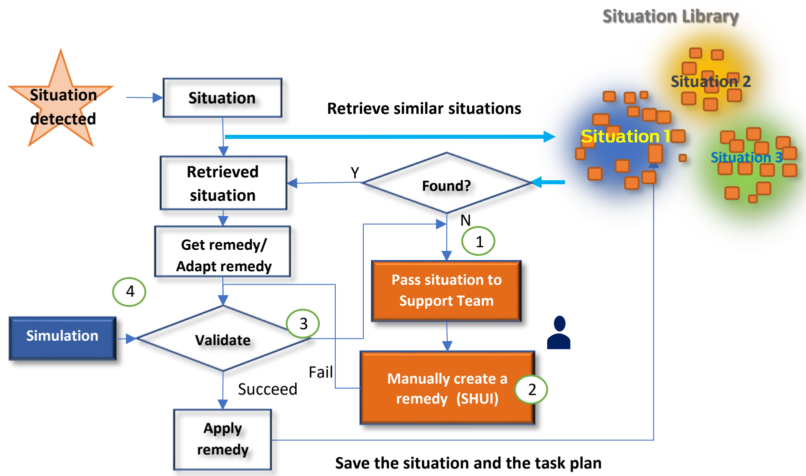

Fig. 3. Situation handling life cycle process

Fig. 4 is an example of the SHUI interface. The left panel (Fig. 4 ③) displays the real-time *Task* execution that is identical to what is in VSA (Fig. 1 ①). The lower-middle panel (Fig. 4 ①) displays the situation context. The support specialist can see exactly what is happening and what has happened at the vehicle remotely. The right panels are pallets that the support specialist pick, drag and drop "*Tasks*" and "remedy actions". The upper-middle panel (Fig. 4 ②) shows the revised **Remedy** and the "submit" button will send the revised **Remedy** to VSA to repair the plan.

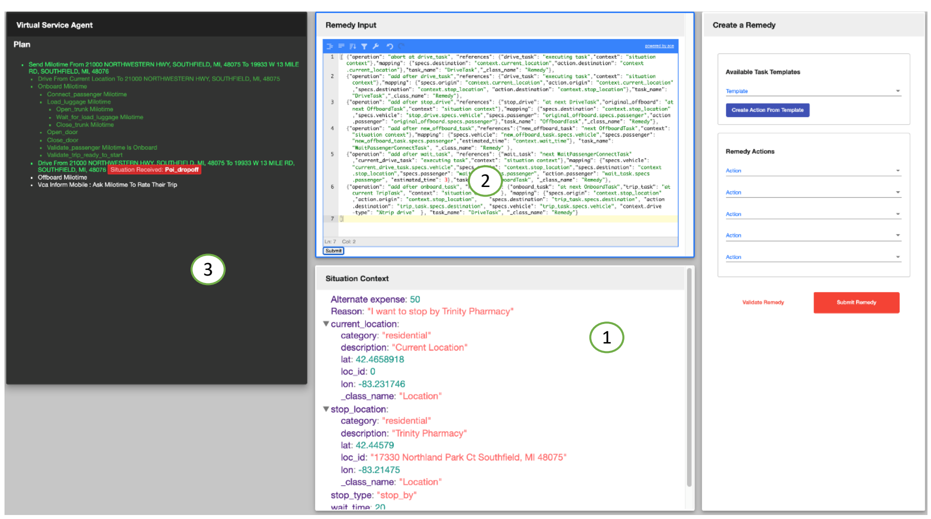

Fig. 4. Situation Handling UI

5 Illustrative Examples

The following are two examples to illustrate how situation handling works.

Example 1: The story of "Window Leak"

[*It starts raining. Passenger Annie saw water seep into the cabin. The window is not fully closed.*]

ANNIE: The water is getting in.

[*The vehicle checks the window, one of them is open.*
The vehicle sends a command to the control unit to close the window. However, the window is not closed.
Vehicle realizes that the window is in a malfunction.
Vehicle recalls a case that the window that could not get closed because the window glass was blocked by a twig.]

VEHICLE: Is there something that blocked the window glass?

ANNIE: Yes, looks like it is jammed

[*The vehicle rolls down the window halfway and the rider cleaned a foreign object that jammed the window. The vehicle rolls up the window again and the window is closed this time.*]

Here is what to happen: Passenger Kelly gets in the car and the vehicle starts the journey.

It starts to rain. However, water seeps through the window, and water drops onto Kelly.

"It is raining and it is wet here", Kelly claimed. A wet-in-cabin *Situation* is generated. The Logics of the *Situation* is:

```
"logics": {
    "window_broken": "vda.checking_window",
    "weather": "weather.current_weather",
    "wetness": "chat.wetness"
}
```

"window-broken" – It calls the vehicle agent to check if any window is broken;

"weather" – from the weather agent, it returns current weather condition;

"wetness" – it initiates a dialog with the passenger to obtain the following information: wherein the cabin is wet (seat? floor? on the person?).

The above information feeds into the Context of the *Situation*.

VSA finds a similar *Situation* from its *Situation* case library that has the following remedy:

```
"add close-window task"
"add confirm-problem-solved task"
```

The "close-window" *Task* is sent to the vehicle, and the vehicle sends a "close-window" command.

The "confirm-problem-solved" *Task* will trigger a dialog using the dialog agent. It returns the confirmation and related response in the form of context.

Unfortunately, assuming, the confirmation is negative. The water is still pulling in. A new "window-fail-to-close" *Situation* is created with all the current context information.

The Logics under this *Situation* is:

```
"logics": {
  "close_window": "vda.close_wdw_status",
  "window_malfunc": "vda.wdw_malfunc_detect",
  "window_broken": "vda.broken_wdw_detect"
}
```

In the above Logics, the "close_window" context is already filled from the previous situation handling process. Therefore, the context is carried over.

Assume we have the following contexts (in addition to all other contexts we have had) after applying the Logics:

```
"context": {
    "close_window": true,
    "window_malfunc": false,
    "window_broken": false
}
```

A similar *Situation* was found that has Remedy:

```
"add confirm-passenger task: window-is-jammed"
```

The answer populates the Context. Assume that the Context is: `"window-is-jammed": true`.

A new similar *Situation* "window-is-jammed" is found and the remedy is:

```
"add open-window task"
"add request passenger task: remove foreign obj"
"add close-window task"
"add confirm-problem-solved task"
```

Assuming the final confirmation is positive, and the *Situation* is resolved. The newly logged *Situation* and the history will be saved to the *Situation* case library and Task case library. In case the final confirmation is negative, and the agent could not find a relevant *Situation*. In that case, VSA may send the *Situation* to SHUI, and human intervention will be called to resolve the *Situation*.

Example 2: The story of "Pharmacy"

[*Passenger Joe went on a business trip. He rides in a vehicle towards the hotel. He passed by a pharmacy and realized that he can pick up a prescription there.*]

JOE: Could you stop by that pharmacy?

[*The vehicle requests the Map Agent to find a pharmacy that is on the way to the hotel. The vehicle shows the map location of a pharmacy on the screen in the vehicle.*]

VEHICLE: Do you want to go to this pharmacy?.

JOE: No, I'd like to go to the one we just passed. [*Joe only wants to go the pharmacy he just saw.*]

[*The Map Agent presents more nearby pharmacies on the map on the screen.*]

VEHICLE: How about these?

[*Joe points to the one he wants to go on the touch screen.*]

VEHICLE: Will you come back and continue your trip?

JOE: Yes.

VEHICLE: How long should I wait?

JOE: Maybe 10 to 15 minutes.

VEHICLE: I will wait for you at the front door of the store in 10 minutes.

[*The vehicle turns around and drives to the pharmacy.*
The vehicle offboards Joe at the pharmacy.
10 minutes later, the vehicle will be back to resume the trip to the hotel.]

Here is how this *Situation* proceeds in VSA:

The Dialogue Agent posts a "`POI_dropoff`" *Situation* (POI - point-of-interest) on the Situation Queue.

When VSA receives the "`POI_dropoff`" *Situation* on the Situation Queue, it attempts to handle the *Situation*.

The *Situation* Header looks like this:

Situation Name: `POI_dropoff`
Task: `Drive_task`
Context: {
  current_location: location . . . ,
  stop_location: location. . . ,
  stop_type: `"stop_by"`,
  wait_time: `15`
}

The situation handling finds a previous "`POI_dropoff`" *Situation* in the case library. The Context of the retrieved old *Situation* has "**stop_type**" of "`final destination`", which means the passenger would choose the "stop-location" as her final destination, she would not continue her original journey. The final destination of the trip was changed to the "**stop_location**" of the *Situation*, defined in the Context. The retrieved *Situation* has three *remedy actions* in the **Remedy**:

```
[
 {"operation": "abort at drive_task"...},
 {"operation": "add after current_drive_task"...},
 {"operation": "modify at next_offboard"...}
]
```

The **Remedy** is:

1. abort the current `drive_task`;
2. add a `drive` task to the **stop-location**;
3. modify the `offboard_task` so that the offboard location is changed to the **stop-location**.

The final destination of the trip was changed to the new "**stop_location**", defined in the Context.

The new "`POI_dropoff`" *Situation*, however, is different such that the passenger will continue his journey to his original destination. This is defined in the goal of the *Situation*.

When **Remedy** of the retrieved *Situation* was adapted to the new "`POI_dropoff`" *Situation*, it encounters an exception in the validation (Fig. 2 ② , Fig. 3 ④), because the goals of the new *Situation* are different. One of the new goals is that the final destination should be the same as the original destination, instead of the "**stop_location**". This exception is captured and VSA will send the *Situation* to SHUI. A new **Remedy** is created manually in SHUI and is sent back (Fig. 4 ②) to VSA. The new **Remedy** has six remedy actions:

```
[
 {"operation": "abort at drive_task"...},
 {"operation": "add after current_drive_task"...},
 {"operation": "add after stop_drive"...},
 {"operation": "add after new_offboard_task"...},
 {"operation": "add after wait_task"...},
 {"operation": "add after onboard_task"...}
]
```

1. abort the current drive_task;
2. add a drive task (stop_drive) to drive to the "stop-location";
3. add an offboard task at the "stop-location";
4. add a wait task after the offboard task;
5. add an onboard task after the wait_task;
6. add a drive task after the onboard task that drives to the final destination.

Applying this **Remedy**, the new plan passes validation (Fig. 5). The new *Situation* with the revised **Remedy** is saved to the *Situation* case library so that next time, similar *Situation* will be handled without human intervention.

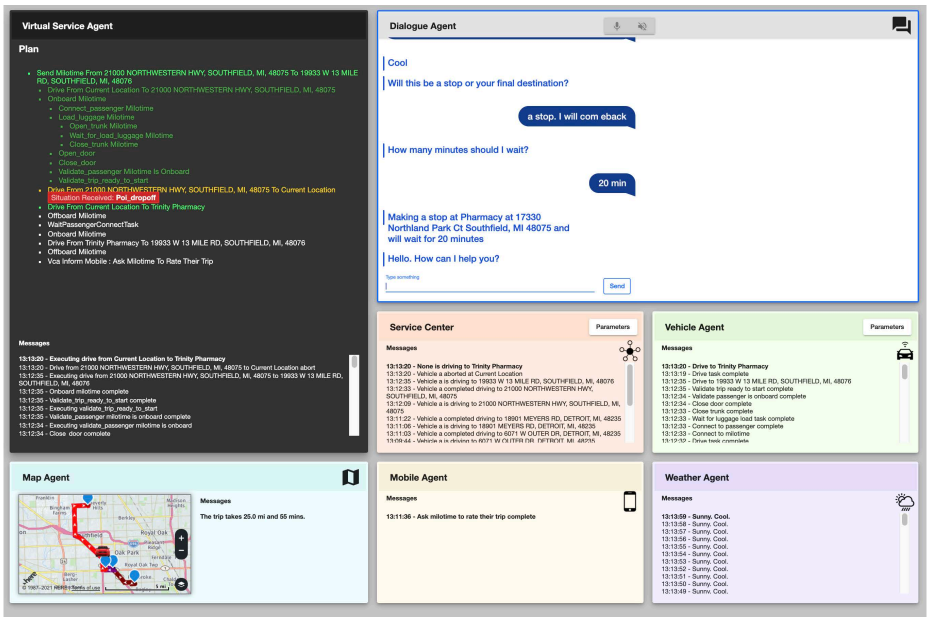

Fig. 5. The repaired plan

6 Conclusions

This paper focuses on solving planning problems for a service agent who faces possibly unlimited *Task* and *Situation* types, with additional context variations, in the real-world. Hard-coding domain-specific knowledge in such a system does not scale. This paper introduces a comprehensive solution that illustrates the possibility of adopting generic structures for tasks and situations, and completely embedding problem-solving knowledge in executed cases, both for task planning and situation handling. The cases can be reused to solve similar new problems. It enables the system easily expandable by continually injecting new *Task* plan cases and *Situation* handling cases.

References

1. Aeronautiques, C., Howe, A., Knoblock, C., McDermott, I.D., Ram, A., Veloso, M., Weld, D., SRI, D.W., Barrett, A., Christianson, D., et al.: PDDL— The Planning Domain Definition Language. Tech. rep., Technical Report (1998)
2. Aha, D.W.: Goal reasoning: Foundations, emerging applications, and prospects. AI Magazine **39**(2), 3–24 (2018)
3. Bergmann, R., Wilke, W.: On the role of abstraction in case-based reasoning. In: European Workshop on Advances in Case-Based Reasoning, pp. 28–43. Springer (1996)
4. Chien, S.A., Knight, R., Stechert, A., Sherwood, R., Rabideau, G.: Using iterative repair to improve the responsiveness of planning and scheduling. In: AIPS, pp. 300–307 (2000)
5. Cox, M.T., Muñoz-Avila, H., Bergmann, R.: Case-based planning. Knowledge Engineering Review **20**(3), 283–288 (2005)
6. Dannenhauer, D., Munoz-Avila, H.: Goal-driven autonomy with semantically-annotated hierarchical cases. In: International Conference on Case-Based Reasoning, pp. 88–103. Springer (2015)
7. Fikes, R.E., Nilsson, N.J.: STRIPS: A new approach to the application of theorem proving to problem solving. Artificial intelligence **2**(3-4), 189–208 (1971)
8. Gerevini, A., Serina, I.: Fast plan adaptation through planning graphs: Local and systematic search techniques. In: AIPS, pp. 112–121 (2000)
9. Ghallab, M., Nau, D., Traverso, P.: Automated Planning and Acting. Cambridge University Press, USA (2016)
10. Hammond, K.J.: Case-based planning: Viewing planning as a memory task. Academic Press (1989)
11. Kambhampati, S., Hendler, J.A.: A validation-structure-based theory of plan modification and reuse. Artificial Intelligence **55**(2-3), 193–258 (1992)
12. Leake, D., Jalali, V.: Context and case-based reasoning. In: Context in Computing, pp. 473–490. Springer (2014)
13. Nau, D.S., Au, T.C., Ilghami, O., Kuter, U., Murdock, J.W., Wu, D., Yaman, F.: SHOP2: An HTN planning system. Journal of artificial intelligence research **20**, 379–404 (2003)
14. Sacerdoti, E.D.: Planning in a hierarchy of abstraction spaces. Artificial intelligence **5**(2), 115–135 (1974)
15. Scholnick, E.K., Friedman, S.L.: Planning in context: Developmental and situational considerations. International Journal of Behavioral Development **16**(2), 145–167 (1993)
16. Tate, A.: Generating project networks. In: Proceedings of the 5th international joint conference on Artificial intelligence-Volume 2, pp. 888–893 (1977)
17. Van Der Krogt, R., De Weerdt, M.: Plan repair as an extension of planning. In: ICAPS, vol. 5, pp. 161–170 (2005)
18. Xiao, Z., Wan, H., Zhuo, H.H., Lin, J., Liu, Y.: Representation learning for classical planning from partially observed traces. arXiv preprint arXiv:1907.08352 (2019)
19. Yang, Q.: Formalizing planning knowledge for hierarchical planning. Computational intelligence **6**(1), 12–24 (1990)
20. Zhuo, H.H., Muñoz-Avila, H., Yang, Q.: Learning hierarchical task network domains from partially observed plan traces. Artificial intelligence **212**, 134–157 (2014)