“Drink me”: Handling Actions through Planning in a Text Game Adventure

LUCIANA BENOTTI
Technical University of Madrid, Spain / Free University of Bolzano, Italy
Luciana.Benotti@stud-inf.unibz.it

ABSTRACT. The general aim of this work is to investigate how the addition of state-of-the-art reasoning capabilities can be useful in a dialogue system. In particular we will work with FrOz, a dialogue system that was developed at the University of Saarbrücken to explore this idea. FrOz is a text adventure game that uses Description Logics to codify a given game state. It is assisted by a theorem prover (RACER) for inference.

In this paper we discuss how to add a planning step to FrOz’s actions module. The actions module is in charge of executing commands indicated by the player whenever the specified preconditions are satisfied in the current state of the game. The point of adding planning capabilities to the actions module is to increase the flexibility of the dialogue system. To achieve this aim we will use a general purpose planning system, Blackbox, that implements modern planning techniques.

This work was partially carried out in the context of the research project “Técnicas de Inteligencia Computacional para el diseño e Implementación de Sistemas Multiagentes” (COD 04/E062) of the Universidad del Comahue, Argentina.

1 Planning in Dialogue Systems

Planning has been used to handle different aspects of a dialogue system for several decades now (Allen 1994). For example, plan-based models are suitable for recognizing speech acts performed by the user of a dialogue system, for inferring her goals, and for cooperating in their achievement (Perrault and Allen 1980). Planning techniques also have been used in natural language generation: when the system needs to convey large amounts of information using multiple utterances, the organization of the information into utterance-size units is often viewed as a planning process (Reiter and Dale 1997).

The planning assistant TRAINS (Ferguson, Allen, and Miller 1996) is a classical example of the use of planning in a dialogue system. The aim of TRAINS is to help the user to solve routing problems in a transportation domain. In such environment, the human and the computer must work together in a tightly coupled way to solve problems that neither of them could manage alone. System and user collaborate in order to build a suitable plan for solving the problem at hand. Then, planning techniques are used to validate the feasibility of the plan that is being constructed, infer user goals and cooperate in their achievement. However, Ferguson, Allen and Miller report that, according to their experience in TRAINS,
and other dialogue systems, traditional planning (finding courses of action from an initial situation to a goal) turns out not to be suitable. To start with, the initial state is usually incompletely specified because of changing conditions or simply because it is too huge to represent. Similarly, the goals of the plan are also poorly specified. Not only they change over time as the user’s concerns and preferences change, but also they typically cannot be extracted and codified in a way suitable for automatic processing. Hence the planning process needs to be tailored and specially implemented for each particular dialogue system.

In our paper we identify a case that can be directly handled by traditional planning, and we show how it is possible to take advantage of a state-of-the-art planning system to solve it. In the task we are going to tackle, initial states, goals and available actions can be completely specified and an off-the-shelf planner can be used to enhance the dialogue system capabilities. The dialogue system we are going to work with is the text game adventure FrOz developed at the University of Saarbrücken (Koller, Debusmann, Gabsdil, and Striegnitz 2004).

In FrOz, all the interaction between player and game is done in natural language. The game can understand commands the player presents as English sentences which verbalize actions that she (the player) wants to execute in the game world (such as “Open the door with the key”). After receiving a command, FrOz verifies if such an action can effectively be executed, checking whether its preconditions hold in the game world. In this case, the game world is updated according to the action effects. Otherwise, the action fails.

It is exactly this behaviour what we want to modify. In certain situations, the player specifies an action assuming that either all preconditions are satisfied, or that they can be easily satisfied by performing trivial additional actions. Let us consider, for example, the situation where the game has just described that a key is lying on a table in front of the player in a room with a locked door. Then the player might input the command, “Unlock the door with the key”, which the current version of FrOz will fail to execute because the player is not actually holding the key (the unlock-with(Object Key) action has the precondition instance(Key inventory-object)). That is, the player is obliged to enter a command like “Take the key and unlock the door with it” for the action to succeed. In such cases, a collaborative dialogue system would try to fill the gaps in the input received from the player, trying to guess the missing actions that the player is assuming to be too obvious to specify explicitly. By doing so, the system would free the player from the nuisance of specifying simple extra actions necessary in order to meet the preconditions.

We would like FrOz to be able to compute autonomously the sequence of actions that should be executed in order to get from the world state where the action fails to the state in which this action can be executed (i.e., the state where all the action preconditions hold), and decide if this sequence is ‘simple enough’ to be executed automatically, even if the player has not specified it in detail. In order to determine which is the proper sequence of actions, a planning step is needed. In this paper we will discuss how a general purpose planner such as Blackbox (Kautz and Selman 1999) can be used to help FrOz do exactly this.
2 FrOz

As a dialogue system, FrOz’s general architecture follows a standard pipeline (Bernsen, Dybkjaer, and Dybkjaer 1997) composed by six modules. In such architecture, depicted in Figure 1.1, a cycle of input-output in the game can be described briefly as follows. First, the player’s input is parsed by the parsing module. This yields a semantic representation specifying the action that the player wants to execute, and also describing the objects that this action involves. Next, the object descriptions are resolved to individuals of the game world by the reference resolution module obtaining a ground term that specifies the action intended by the player. During the third step, the actions module looks up this action in an action database, checks whether its preconditions are met in the world, and, if so, updates the world state with the effects of the action. The changes introduced in the current situation are then reported to the player through natural sounding English text, which is automatically generated by the remaining three modules: content determination, reference generation and realization. FrOz implements state-of-the-art techniques from computational linguistics for each one of these modules.

The game can be instantiated with different scenarios. The functionality offered by the different modules is shared by all scenarios, while each scenario has its own action database and knowledge bases where it codifies its specific characteristics. FrOz already defines some scenarios like the “Space Station”, the “Fairy Tale Castle”, etc.

FrOz uses Description Logic (DL) (Baader, Calvanese, McGuinness, Nardi, and Patel-Schneider 2003) knowledge bases to codify the information concerning the state of the
game. A DL knowledge base is a pair \( (T, A) \) where \( T \) is a set of definitions and \( A \) a set of assertions. FrOz’s knowledge bases are accessed by almost all modules in the pipeline (see Figure 1.1) via queries sent to the RACER reasoner (Haarslev and Möller 2001).

In particular, underlying the system there are two knowledge bases, which share a set of common definitions (the T-Box) and differ only in their set of assertions (the A-Boxes). The common T-box defines the key notions in the world and how they are interrelated. Some of these notions are basic concepts (such as object) or properties (such as alive), directly describing the game world, while others define more abstract notions like the set of all the individuals a player can interact with.

The A-Boxes specify the kind of an individual (for example, an individual can be an apple or a player). Relationships between individuals in the world are also represented here (such as the relationship between an object and its location).

One of the knowledge bases (the world A-Box) represents the true state of the world, while the other (the player A-Box) keeps track of the player’s beliefs about the world. The assertions listed in the player A-Box will typically be a strict subset of the assertions in the world A-Box because the player will not have explored the world completely and therefore will not know about all the individuals and their properties. It may happen, however, that some effects of an action are deliberately hidden from the player; for example, if pressing some button in a room has some effect in another room which the player cannot notice. In this case, the player A-Box may actually contain information that is inconsistent with the world A-Box.

These possible inconsistences need to be taken into account when integrating FrOz and the planner Blackbox. We will return to this in Section 4. Now, let us focus in the actions module in FrOz, the module that will interact with the planner.

### 2.1 The Actions Module

As we have already mentioned when we described FrOz’s architecture, the actions module receives a ground term that specifies the action intended by the player. For example, if the input is “Take the key”, the ground term received by this module will be:

\[
take(key1)
\]

where \textit{key1} is the unique individual in the player A-Box that resulted from the resolution of the determiner “the key”.

Given this representation of the action intended by the player, the actions module is responsible for finding the appropriate entry in the actions database, which specifies the action preconditions and effects. This database can be seen as the codification of the ‘instructions’ that guides the actions module in fulfilling its task. The actions module uses RACER to perform the necessary inferences in order to follow these instructions. The database is specified in a STRIPS-like format (Fikes, Hart, and Nils 1972) and it divides the effects of an action into those that modify the world A-Box (effects) and those that modify the player A-Box (player beliefs) when the action is executed.

An example of an entry in the actions database is given below:
The term $X$ in the action representation shown above is a variable that gets bound to the actual argument that the resolution module computed. In our previous example, $X$ would be bound to the constant key1, and thus the preconditions and effects of the operators will become ground terms.

It is important that we grasp how actions are dealt with in FrOz if we want to understand how to integrate planning capabilities in the actions module. Let us see in detail how to read the specification of the action take when applied to our example.

The command “Take the key” issued by the player requires that the key is accessible to the player (instance(key1 accessible)), that it is small enough to be taken (instance(key1 takeable)) and that it is not carried by the player already (not(instance(key1 inventory-object))). When this command is executed, the key becomes an object in the player’s inventory (instance(X inventory-object)) and it is no longer located where it used to be. This last effect includes an expression that requests RACER to return the individual in the world that represents the location of the key (individual-filler(key1 has-location)). A RACER expression is embedded in the action specification when the action cannot be specified completely in advance because it depends on the current state of the game (as is the case for most interesting actions). Finally, for this action, the effects on the player beliefs are identical to the effects on the world state.

Remember our example in Section 1 where the player tries to unlock a door with a key that she is not holding. FrOz current actions module will fail to execute the action unlock-with(door1 key1) because its precondition instance(key1 inventory-object) is not satisfied; the action we have just explained, take(key1), needs to be executed first. If we want the actions module to find autonomously the sequence of required actions that should be executed to bridge the gap, we need to enhance FrOz with planning capabilities.

## 3 Blackbox: an Off-the-shelf Planner

Blackbox (Kautz and Selman 1999) is a planning system that works by converting planning problems into Boolean satisfiability problems, and then solving them with a variety of
satisfiability engines. The front-end employs the graphplan system (Blum and Furst 1995) while the back-end role can be played by different satisfiability engines, allowing the use of the engine that is best suited for a particular type of problem.

Blackbox works by fixing the length of the plan in advance and iteratively deepening it. This behaviour makes it particularly well suited for our needs. To begin with, it finds optimal plans (minimal in the number of actions). Optimal plans are crucial because FrOz cannot force the player to do unnecessary actions. Moreover, Blackbox is extremely fast when searching for short plans, and these are exactly the kind of plans that we need in our framework, as it does not seem sensible to allow too much autonomy to the game. Of course, fast responses are critical for a natural interaction with the player.

The input required by Blackbox are STRIPS-style problems specified in the standard Planning Domain Definition Language (PDDL) (Gerevini and Long 2005). A PDDL specification consists of two parts: the domain and the problem. The longest and more complex of these two is the domain specification that contains a crucial element in any planning domain specification: the actions (with its associated parameters, preconditions and effects). On the other hand, the problem specification is relatively simple and contains the initial state (which describes the state of the world at the beginning of the plan), the goal (which represents the desired state of the world after the plan execution) and the objects (that can instantiate the actions) with their corresponding types.

When Blackbox is invoked with a domain, a problem and a maximum plan length it will return an optimal plan of smaller or equal length than the maximum specified, if such a plan exists. Otherwise, it will report that there is no such plan.

In the next section we will discuss how to generate suitable PDDL specifications so that Blackbox is able to find plans in the context of the game.

4 Blackbox in FrOz

In our redesigned version of FrOz, when an action fails because some of its preconditions do not hold in the world, the actions module will invoke Blackbox as depicted in Figure 1.2. Blackbox input, conforming the format described in the previous section, will include on one hand the domain specification describing the FrOz scenario that is being played. On the other hand, the problem specification will represent the current and the intended state of the game corresponding to the initial state and goal respectively. Also, we will instruct Blackbox to find plans of up to two actions only. Longer plans are probably not useful. Bear in mind that FrOz is only attempting to 'guess' trivial actions that were left unspecified by the user. Let us describe the domain and problem specification in more detail.

Codifying a Domain  As we mentioned, the domain specification is complex, and hence difficult to generate, involving a number of important design decisions. However, as it is independent of the state of the game at a particular moment, it can be generated offline, once and for all for each game scenario. The information required is obtained mainly from
the scenario actions database but we also need to query RACER about definitions in the knowledge bases in order to complete the domain specification.

Let us discuss the main intuitions behind the generation of suitable action specifications. As actions in both FrOz and Blackbox are described using a STRIPS-like format, we can expect that the translation is simple. This is true except for two difficulties. Consider the action `take` we discussed in Section 2.1. It would be encoded in PDDL as follows:

```pddl
(:action take
  :parameters (?x - takeable ?y - top)
  :precondition
    (accessible ?x)
    (no-inventory-object ?x)
    (has-location ?x ?y)
  :effect
    (inventory-object ?x)
    (not(has-location ?x ?y))
)
```

The first obvious difference between the two representations is that action parameters in FrOz are not typed, while Blackbox allows typing. And indeed we want parameters to be typed, because this prevents Blackbox from trying out every action over every single individual in our domain (this would easily lead to a blow up in the plan search space). We can type the parameters in the following way. First, let us call a concept in the game knowledge bases static if it does not appear in any action effect in FrOz’s actions database (in this case, it will clearly never be affected during the game). All other concepts are called dynamic. Now, we say that a parameter $?x$ belongs to the type $t$ if $t$ is a static concept and $\text{instance}(X \ t)$ is a precondition for the action. In this way we avoid the instantiation of the action `take` with individuals that do not satisfy this precondition (moreover, the
precondition is no longer necessary and we can eliminate it from the specification of the action. In cases when no such precondition exists then \(?x\) belongs to the type \texttt{top}.

The second evident discrepancy in the proposed translation is that the action has now two parameters instead of only one. The second parameter (\(?y\)) represents the individual that the RACER expression \texttt{individual-filler(X has-location)} resolves to. But, how can we be sure that Blackbox will instantiate this parameter with the appropriate individual? We just need to tell the planner that \(?x\) has to be related with \(?y\) by the functional role \texttt{has-location} adding the precondition \((\texttt{has-location ?x ?y})\) to the action. In general, the embedding of RACER expressions in action specifications is the most complex issue we have to deal with and our approach is to add a parameter and its corresponding precondition for each such expression.

Having explained this, the required encoding can be directly obtained from the representation of the action \texttt{take} in FrOz’s actions database$^{1}$.

**Codifying a Problem** The problem specification clearly depends on the state of the game and on the input of the player at a particular moment. Hence, this specification should be automatically generated on-line during the execution of the game.

A problem specification consists of three parts. The first one, is the definition of the objects in the problem, with their corresponding types. This can be easily handled by asking RACER which are all the individuals in the knowledge base and their corresponding types. The second part is the \textit{initial state}, a description of a particular game state where an action fails because some of its preconditions do not hold. RACER can tell us which are the dynamic concepts each object belongs to, and we just have to assert this information in the specification. The last part is the \textit{goal} of the planning problem. Let us analyse what this goal should be. If we want the player to be able to execute an action she was not able to execute before, we need the preconditions of such an action to hold. Then, the goals of our planning problem will be the preconditions of the action that the player wants to execute. This information can be obtained instantiating the preconditions of such action with the objects that the player is intending to manipulate.

In order to define the first two elements we need to query RACER about the objects and their properties in the knowledge base. However, FrOz has two A-Boxes (the game’s and the player’s) and we need to decide which one can give us the information we need. It seems natural to choose the player’s because we do not want Blackbox to return plans including actions that the player is not aware she can perform. However, we should remember that this knowledge base may contain information that is inconsistent with respect to the current state of the game. So it is possible that Blackbox actually returns a plan that cannot be executed over the game world. But even in this case the plan is useful because it leads the player to find out about her misconception about the world. To clarify this point, let us return to the example introduced in Section 1, where the player tries to unlock a door

\footnote{The actual encoding needed for Blackbox is slightly more complicated because we have to deal explicitly with the Closed World Assumption (Blackbox does not do it automatically), but the main intuitions are clear from the example we described above.}
with a key. The key was originally lying on the table, but without the player knowing, it is now in possession of a cat (who has the surprising ability to appear and disappear at will!). As a consequence, the key is on the table in the player’s knowledge base, but in the game world the cat has it. With the added planning capabilities, FrOz would decide to take the key and unlock the door with it. But this sequence of actions will fail (when the preconditions are checked by the actions module) because the key is no longer accessible, and this situation will be informed to the player. If, instead of finding a plan according to the player’s beliefs, we would have planned using the world knowledge, FrOz would have automatically taken the key from the cat (for example, by using the steal action) and opened the door for the player, while the player is not even aware where the key actually is. This is clearly inappropriate because we only want FrOz to take actions for the player if we can be sure that these actions agree with the player intentions.

Given the domain and problem specifications and a maximum length for a plan, Blackbox will be able to find the sequence of actions required in order to get from the state where the action fails to the state in which this action can be executed according to the player’s beliefs, in case such a sequence exists. If this sequence is simple enough (we will return to this issue in Section 6) then FrOz will execute it and finally it will execute the action input by the player.

Given this setup, Blackbox’s performance is impeccable. For the world domains provided with FrOz (around 20 actions schemes and 30 individuals that instantiate around 60 actions), it only takes the planner a couple of milliseconds to find a suitable plan or to answer that there is none. In order to check the scalability of our approach we tested Blackbox with up to 6000 instantiated actions, but even for this huge problem Blackbox still takes less than a second to return a plan or to say that there is none. Blackbox performance does not seem to be a problem in our setup.

To round up our discussion and make things concrete, let us discuss a worked out example in full detail.

5 Alice and the Bottle

Suppose we developed a FrOz’s scenario based on “Alice in Wonderland,” and Alice is now in the rabbit-hole.

The game has just described that there is a little bottle on a table in front of Alice. Around the neck of the bottle there is a paper label, with the words ‘DRINK ME’ beautifully printed on it in large letters. Then, Alice (the player) might input “Drink the bottle”. Without the work described in this paper, FrOz would answer such a demand with a negative response: “You can’t do this! You do not have the bottle!”. Actually, two preconditions in the action \texttt{drink(X)} have failed as the player should have the bottle in her inventory (\texttt{inventory-object bottle1}) and it should be uncorked (\texttt{uncorked bottle1}) for the action to succeed.
In the new version of FrOz this would not happen. Instead Blackbox would be invoked and a suitable plan would be found. The specification used during the call would include the “Alice in Wonderland” domain, the current state of the game and the goal. The goal would include the preconditions for the action `drink(bottle1)`:

```lisp
(:goal
   (inventory-object bottle1)
   (uncorked bottle1)
)
```

Given this goal, the plan output by Blackbox would include two actions:

```
Begin plan
1 (take bottle1 table1)
2 (uncork bottle1)
End plan
```

The actions ‘take the bottle’ and ‘uncork the bottle’ can be performed, in that order, from Alice’s current state in the game. With this information FrOz is able to execute these two actions on its own and respond: “You have the bottle. The bottle is uncorked. You drink the bottle.”, a much more friendly and natural answer than FrOz’s original reply.

6 Discussion

The ideas presented in this paper are still ongoing work, and there are many more interesting research directions to investigate.

To begin with, we will discuss how well our planning task is handled by Blackbox. Blackbox’s behaviour can be tailored keeping in mind that FrOz is a computer game. In particular, it is very important that plans returned by Blackbox are both optimal and short. Optimal plans (i.e., plans containing no superfluous actions) are needed because otherwise we risk executing actions the player would not have thought of performing. Moreover, we need short plans because we do not want the planner to solve the game for the player! In this respect, we can easily instruct Blackbox to look only for plans with up to two actions. As we mentioned, given our setup, Blackbox performance is not a issue. However, there is room for improvements. In some cases, our new version of FrOz will not be able to come up with a plan, even if one exists, because the encoding described in Section 4 is incomplete. This is due to the fact that Blackbox input language is less expressive than the language supported by RACER. As arbitrary queries to RACER can be embedded in FrOz actions databases, we cannot expect to be able to cover them in full generality. Nonetheless, the
The proposed encoding does improve FrOz behaviour in many common cases. Maybe in the future, planners will be able to handle more complex cases within the time restrictions imposed by an interactive system such as FrOz.

Once the planner outputs a plan, there is one additional issue that should be taken into account. We want the game to perform actions for the player only if the game can assume that the plan is sufficiently ‘simple’. How can we guarantee this? One possibility is the following, given a game scenario some actions can be considered minor or trivial, while others cannot. Putting Humpty Dumpty back on the wall is definitely not a trivial action (after all, not even all the king’s horses and all the king’s men were able to do it!). This classification of actions into essential (which can only be performed explicitly by the player) and minor (which can be executed via planning) should be specified by the scenario designer; and the planning domain would include only those actions that are specified as minor. We could also think of making this distinction dynamic, so that actions which have been already performed at least once by the player can now be considered minor.

The work we discussed has also some interesting consequences for generation. The generation component could be enhanced so that it will render differently the case where the changes it is reporting over the world were explicitly indicated by the player, and the case where they were caused by actions automatically performed by the game. For example, if the player inputs “Take the bottle, uncork it and drink it” FrOz normal reply will be “You have the bottle. The bottle is uncorked. You drink the bottle.” However, if the player’s input was “Drink the bottle” and the other two actions are inferred by the planner as in our example in Section 5, a more natural (and informative) answer would be “You drink the bottle [taking and uncorking it first].”

7 Conclusions

One of the aims in the original FrOz design was to analyse the feasibility of integrating a state-of-the-art reasoning system like RACER, and proving that it could be useful for providing the reasoning capabilities that a dialogue system requires. Our paper discusses the addition of a different kind of inference: planning. And also this time our goal is to use an off-the-shelf system to perform this task. We believe that finding the way to take advantage of current technologies is an important issue if our goal is to design a generic dialogue system.

The integration of a generic planner we have described can be useful for other interesting extensions of the game. For example, offering ‘hints’ to the player during the game execution, or answering player’s questions. Such extensions will be the topic of our future research. But the next issue in our research agenda is evaluation. We should empirically test whether the plans we obtain from Blackbox in our setup are indeed useful in real game situations. This can only be verified by compiling and analysing a corpus of actual interaction with the new version of FrOz.
Bibliography


