

GUIDE

Daniel Dionne, Salvador de la Puente, Carlos León, Raquel Hervás, Pablo Gervás

Universidad Complutense de Madrid

Madrid, Spain

{dionnegonzalez, neo.salvador}@gmail.com,
{cleon, raquelhb}@fdi.ucm.es, pgervas@sip.ucm.es

Abstract

This document presents the implementation details for the proposed system by the NIL Research Group in the GIVE Challenge. It tries to show all the relevant information that could be useful for inspecting how this system has been designed and programmed. Also, how the GIVE Challenge could be improved in future versions is discussed.

1 Introduction

When we decided to take part in the GIVE challenge¹ for the first time, we were sure that our solution would include Natural Language Generation: we wanted our virtual guide to communicate with the user as a human guide would do. For this reason, the guide not only had to work with a “human” abstraction of the world, but also, when building instructions, it had to consider all the information. Explicit information as well as inferred information.

This text is an essay about the development of our solution, which we call **GUIDE**. It is oriented to the hosting team as well as the rest of participants, which we assume are familiar with the concepts of the challenge. In other case, we recommend reading the previous paper on this work (Dionne et al., 2009), from the same authors, as an introduction to the aspects that follow.

While developing GUIDE, a framework for virtual guide building was being designed. The result of our research can be seen on figure 1, which will also serve for better understanding of this section.

The architecture has four differentiated parts: **instruction tree**, **instruction planner**, **disambiguation** and **alerts**; each part having one of

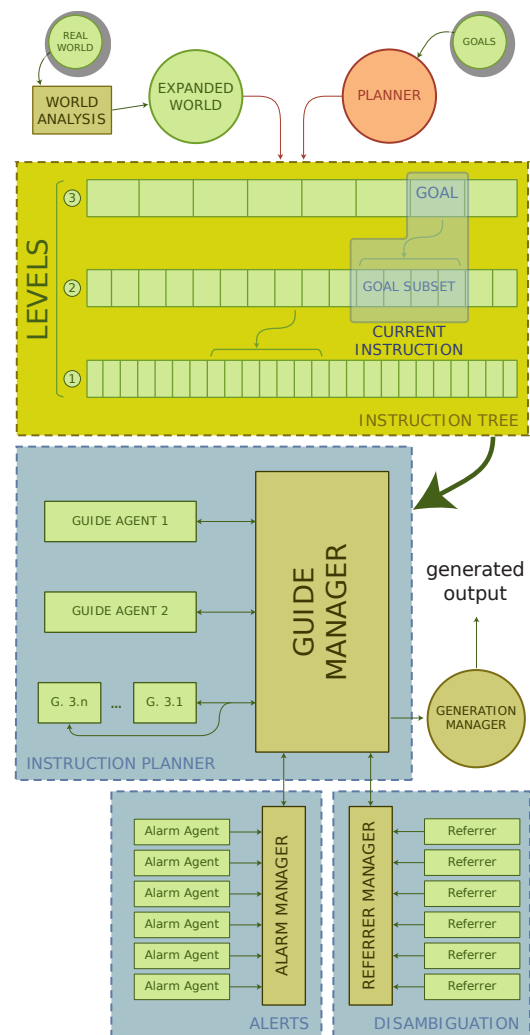


Figure 1: General design.

¹The GIVE challenge is (Koller et al., 2007) part of the ENLG09 celebrated in Athens. The challenge’s goal is to design a virtual guide (VG) that tells the user a set of steps to follow in order to accomplish a task.

more modules. Section 2 briefly describes our development process through each of these parts.

2 System Overview

Our starting point was the development of a logic structure of the world, closer to the human ideas of indoor enclosures, such as rooms, hallways, lobbies, intersections, etc. During this stage, which we call *world analysis*, we build an enriched model of the environment, the *expanded world*, built on top of the simple structure of the GIVE framework. The expanded world contains a list of rooms (or spaces), each of the populated with some information (like type, perimeter, number of walls, corners...). It also has an interconnected graph of rooms, a list of “invisible” reference objects such as doors or intersections, and some methods to determine in which room the user is at, if the user has already been there, etc.

The goal of having a more human representation of the world is to simplify the set of instructions that the GIVE planner offered. In general terms, solving a plan consisted on doing something in the current room and going to the next room, then repeat. This can be compared to an example of a basic plan the GIVE planner would build, such as “step forward, turn right, step forward, step forward, turn left...”. It seemed obvious that most of the steps were irrelevant, so we only considered steps that had some manipulation or where the user switched between rooms, being the rest of the plan ignored. This way, the original plan could be summarized into a new, shorter and simpler plan. Nevertheless, thinking that the original plan should be used at some point, we decided to design our own “higher level” plan on top of the original. This way, each instruction of the new level would keep a link to the related or summarized instructions on the lower level. For example, with the expanded world, a group of the basic plan’s movement instructions that made the user go from one room to another were condensed into one instruction, grouping from the last manipulating or room switching instruction to the next one.

As an example, consider Figure 2, and the difference between a plan such as “turn left, step forward, turn right, step forward, step forward, step forward, turn right, step forward, step forward, turn left, step forward...” and the instruction “exit the room”, which is simpler, faster to accomplish, and summarizes all the information of

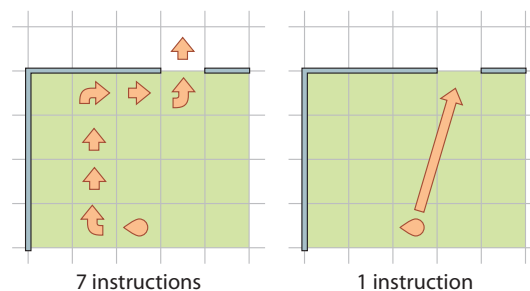


Figure 2: A comparison of a step by step plan versus a human readable plan like “Walk out the door”. Note the difference in the number of instructions given.

the original plan.

This new level made us think about other levels, on a higher (or lower) level of abstraction than the previously explained, so we decided to generalize this solution with what we call *instruction tree*, where each node represents an instruction, and the children are the grouped instructions of the lower level. We will further explain this structure in Section 3. For now we will say that the overall idea is the interpretation of the width and height of the tree. Width represents the progress through the plan, while height represents the level of detail of each instruction. The lower the level, the higher the detail, the more steps and, usually, the longer it takes to accomplish the task.

So we had a plan, but its structure was much more complicated than a simple instruction list, and navigating through it was not so simple. The *guide manager* is in charge of navigating the instruction tree, deciding when it is necessary to go to a deeper level, or climb to higher one, and when it is time to go to the next step. It is also responsible for deciding where to include the information from the other parts of the system. The different *guide agents* that are shown on the figure translate each instruction to the format required by the *generation manager*, which is the NLG module the guide is using. The set of guide agents and the guide manager are part of the *instruction planner*, which synthesizes the best suited instruction at each point.

Instructions usually involve manipulating elements from the environment. Elements that, because of their properties, can be hidden by other similar elements. In other words, there can be ambiguity. To solve this problem we created the *disambiguation* module, which builds sentences that

refer uniquely and without ambiguity to the desired element. It uses *referrer agents*, each of them in charge of finding certain properties from the referred object in the group of distractors, in order to include or not this property in the final sentence to reduce the complete set of distractors. We will see how these agents work in Section 3.

Finally, the *alerts* module is in charge of proactively look after the user. *Alarm agents* are alarm triggers, strategically placed in specific points of the environment, triggered by environmental conditions or the user's state. It is possible that several alarms are triggered at the same time, in this case the *alarm manager* should determine the preference of the alarms and enrich the sentence through the guide manager, in order to include the relevant information. In our implementation, alarms are placed on GIVE alarm tiles, on distractor sets elements or even on doors that should not be used when exiting a room.

This ends the general vision of the GUIDE system. Section 3 contains additional details for each module.

3 Detailed System

Section 2 contains a brief explanation of each part of the system. In this chapter we will extend this information, explaining the most important details of our implementation.

3.1 World Analysis

As we introduced in the previous Section, this chapter offers a more human abstraction of the virtual environment than what GIVE offered. The first challenge was to group discrete tiles, which defined rooms by their adjacency, in order to recognize the type of rooms depending on their shape or how they were connected to other rooms. The problem identifying space regions is trying to guess how we distinguish between one space region or several independent regions. Consider Figure 3. In (A) the reader might distinguish a single region with shape of the letter H, with exits in the upper left and lower right corners. Nevertheless, in (B), one can observe three independent regions. The only difference is the thickness of the walls.

This is a classic problem in mobile robots development. In these cases, the topographic map is built on sensorial analysis: for example, a sonar obtains a profile of the near environment, and a later analysis groups space into bigger regions.

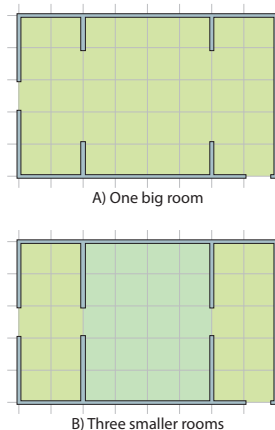


Figure 3: Defining a *distinguishable* space.

Nevertheless, in a virtual environment there are no sensorial sources and either the space regions previously exist or are to be inferred from the original structure. The algorithm 1, developed for this reason, groups space regions of the GIVE environment into rooms.

```

1: cursor ← user position
2: orientation ← user orientation
3: advance cursor facing orientation until it finds a wall
4: start location ← wall location
5: turn right
6: add new unexplored room to the room list
7: while there are unexplored rooms in the room list do
8:   while cursor is not at start location do
9:     advance cursor facing orientation until it finds a corner
10:    if cursor comes up with a concave corner then
11:      turn right
12:    end if
13:    if cursor comes up with a convex corner then
14:      if there is a door at cursor location then
15:        add a door at cursor location
16:        add new unexplored room to the room list
17:      else
18:        turn left
19:      end if
20:    end if
21:  end while
22:  add discovered border to the room
23: end while

```

Algorithm 1: Looking for rooms.

Its operation is simple when considering Figure 4. Its task is to find the walls of a room (A) and then discovering the concave profile of the it. If a convex corner is found, an attempt to follow the original direction is made. If a small gap is found, it is considered a door, but if the gap is big (B), then it is considered a convex corner, and the direction of the exploration changes (C). If the gap is small, the exploration continues (D) after adding

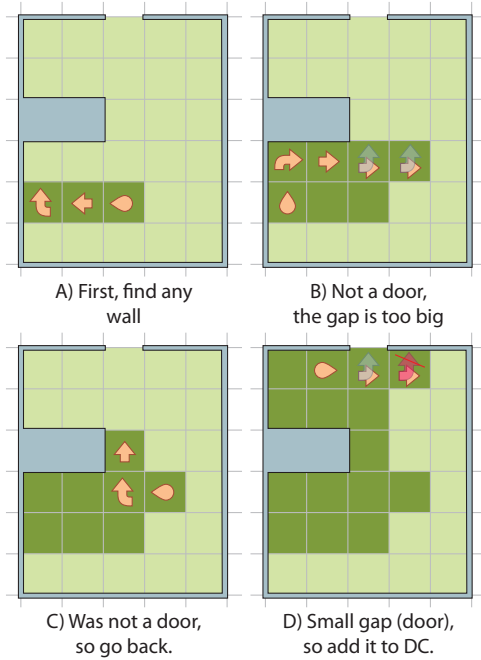


Figure 4: Looking for rooms.

an exit to the room and leaving the other side for further exploration.

In our implementation, while discovering rooms, aid objects such as doors or corners are added, and the adjacency graph is built. In our case, we distinguish between rooms and corridors, as well as certain type of hall intersections, though a further analysis of the adjacency graph would obtain better information from the rooms. This technique is also used in the sensorial exploration mentioned before.

Finally, a filling algorithm explores the remaining space, going from the room's walls to the center.

3.2 Instruction tree

The *instruction tree* is a structure that substitutes the original plan, as a list of instructions, from GIVE. In this tree, nodes represent instructions and the father-children relationship means that the father node somehow summarizes or groups the information contained in the children nodes. As explained in Section 2 and as seen on Figure 5 the instruction tree has a double dimension of information: height represents the progress along the plan, while width represents the level of detail.

How levels are designed and how the tree is built is the developer's task. Our implementation for the GIVE challenge had four levels. The lowest level was the original GIVE plan, which was

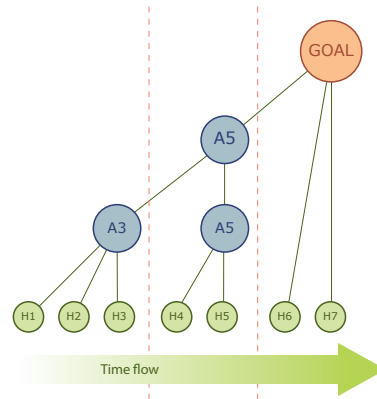


Figure 5: Tree representation of the plan at several levels.

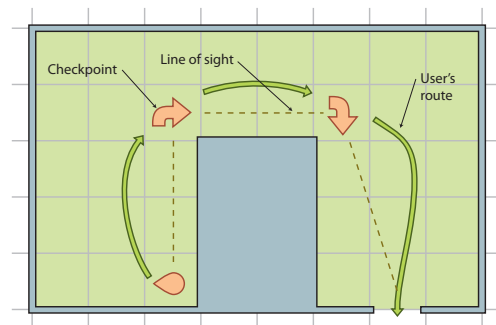


Figure 6: An n-shaped room does not let the user see the exit of the room so VG can guide the user from checkpoint to checkpoint.

hardly used. The second level only had directional changes, in order to manage special situations, such as the one illustrated in Figure 3.3, where the user can't see the goal of the current task and making necessary a more detailed guiding. The third level considered room switching and object manipulations. The last level only had one node with the plan's goal: "take the trophy".

Splitting the plan into these exact levels corresponded to a general planning model that the team developed, and that can be informally summarized as *any plan consists in accomplishing tasks in the current room and then moving to the next task, repeating until the final goal is reached*. Different approaches reach different instruction trees: for example, a further implementation could consider the following planning criteria: *every plan consists on realizing several tasks as fast as possible*. This "hasty planning" tries to minimize the number of steps considering object manipulation in first place in order to reach the goal.

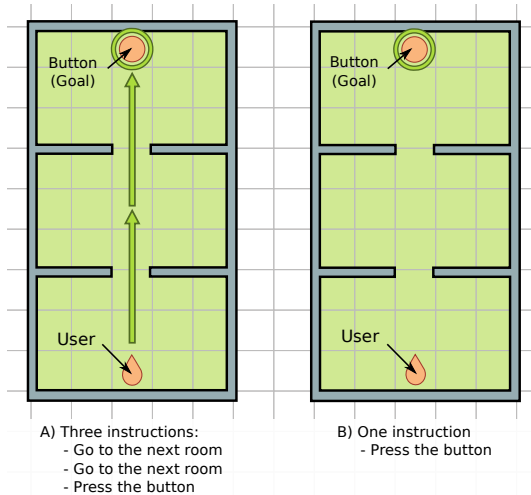


Figure 7: Two different approaches to the same problem: user just has to manipulate the button and he knows where it is because he can see it, but in (A) the plan attends to the maxima “complete tasks in this room and go to the next room” meanwhile in (B) the plan attends to the maxima “reach the goal as soon as possible”.

For this purpose, we have defined five types of high-level instructions: **MovementInstruction** (guides the user from tile to tile), **CheckpointInstruction** (guides the user from a his current position to a checkpoint), **Room2RoomInstruction** (guides the user from room to room), **ActionInstruction** (tells the user to interact with some element) and **GoalInstruction** (subtype of ActionInstruction concerned with achieving the final goal).

Figure 7 compares the same situation, considering both premises. Note that obtained plans are different.

3.3 Instruction Planner

The *guide manager* is the main component of the GUIDE system. Not only it decides how to navigate the instruction tree, but it’s also responsible for using information from other modules to build the next instruction. For these reasons, this part is called *instruction planner*.

Before further explanation about how the instruction tree works, it’s necessary to make a distinction between action and instruction. We define **action** as *anything the user can do that modifies the state of the world* and **instruction** as *an action that the user should perform in order to advance in the plan*. Instructions are defined in terms of preconditions and postconditions. **Preconditions** are conditions that must be satisfied for the instruc-

tion to be performed, and **postconditions** are the conditions that must be satisfied to consider the instruction done.

In general terms, the guide manager tries to explore the tree on its width, though sometimes switching to a lower level is necessary. Starting on any instruction, on any level, the guide manager checks if the actual instruction postconditions have been satisfied. In this case, this branch can be cropped, ascending to the upper level and starting over. If not, the actual instruction preconditions are checked, and if they are satisfied, the guide manager uses an interpreter for that instruction, which we call *guide agent*, to express the instruction’s contained information, adding it to the actual discourse planning. If it’s not possible to descend to a lower level, the instruction tree does not have enough information to guide the user, so a re-planning is needed, making a new instruction tree that will substitute the old one.

The guide manager can check preconditions and postconditions using an action stack and the world’s actual state.

Now we will detail preconditions and postconditions for the instructions in the “*hasty planning*” mentioned in Section 3.2.

MovementInstruction guides the user from tile to tile. The preconditions are satisfied when the current user’s position is the position of starting tile and the postconditions are satisfied when the user’s position is the position of ending tile. Instruction refers both tiles, starting and ending.

CheckpointInstruction guides the user from a certain point to a checkpoint. In Figure , we can see a special shape of room that does not let the user see the exit of the room. In this cases, the VG will guide the user following the shape of the room. We called each of the turns a checkpoint. Now the preconditions are satisfied when user can reach the checkpoint (i.e. there is no obstacles between the user and the checkpoint) and postconditions are satisfied when user can reach the next checkpoint. These instructions refer the current checkpoint and the next checkpoint.

Room2RoomInstruction guides the user from room to room. Preconditions are satisfied when user is inside the starting room and postconditions are satisfied when user

reaches the ending room. It refers the starting and ending room and the entrance the user must use.

ActionInstruction tells the user to interact with some element. Preconditions are satisfied when user can see the element and postconditions are satisfied when the user's last action was an interaction with the element. Obviously, these instructions include a reference to the element the user must to interact with.

GoalInstruction A subtype of ActionInstruction concerned with achieving the final goal.

Consider the simple execution example on Figure 8.

3.4 Disambiguation

Many of the instructions imply object manipulation, and though the planner knows each object and though its logical representation (ID) in the world is unique, the user has to differentiate between that object (the referent) and others with similar properties (the distractors) without using the logical ID. This is necessary even in room switching instructions, as there might be more than one exit in the same room.

Before posting an instruction, the guide manager will need to disambiguate each of the objects that are related to the instruction, using the *reference manager*. The reference manager will try to find the best way of identifying an object, using its unique properties. The goal of the *reference agents* is to enrich the set of properties of an object until the reference manager can refer to that object uniquely, using, for example, Reiter and Dale's algorithm (Reiter and Dale, 1992).

Some of these properties can be directly checked through the world's or the object's state, and others must be inferred from the environment. For example, the center spot in a room is not a visible or tangible object, and finding it requires a non-trivial calculation of the room's shape. Adding it to the references container can help creating simpler and richer sentences. A reference like "the table across the room" can be generated when the listener and the target are in line with the center spot of the room, on opposite sides, independently of where the user is facing. In an indoor environment, architectural elements usually make many inferences possible. Two hallways that intersect make an intersection, two walls make

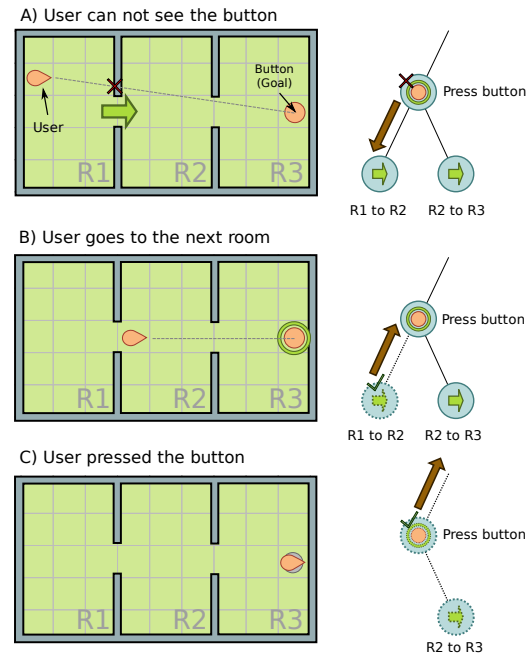


Figure 8: Tree navigation fragment. In (A), the guide manager checks the postconditions for the button manipulation instruction. Since it has not been manipulated yet, it checks the preconditions, to see that the user can't see the button. Then the guide manager descends one level (brown arrow). Now the guide manager tries to execute the instruction that moves the user from room 1 to room 2. The postconditions are not satisfied because the user is in room 1, so preconditions are checked. Since the user can see the exit, this instruction is executed, and the guide manager ascends one level in (B). That branch can be cropped, and the instruction for the button is tried once again. Again, postconditions are not satisfied, since the button has not been pressed yet, but preconditions now are satisfied, now that the user can see the button. The instruction is executed in (C), the branch is cropped, and the guide manager ascends to the upper level.

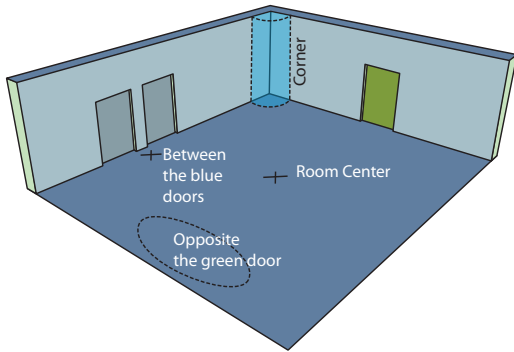


Figure 9: Hidden references in a room.

a corner, etc. and though these elements might not be referenced as they are in the given environment, they should be taken into account. In a similar way, hidden relations discovery can be accomplished. Object alignments or arrangements can be revealed and used for the same purpose. Sentences like “the car in line with these pillars” can be generated. All of these additional high-level concepts and relations between them and low-level world entities are obtained by abstraction over the available representation. We create a family of *reference agents*, each one specialized in identifying candidate disambiguating properties of a different kind. Some of these properties are already explicit in the world representation (colour) and some require a process of abstraction (relations to corners, for instance). Once obtained, they become available as additional properties that may be used to disambiguate references.

What a reference agent does is check a particular referent’s property, and take out from set of distractors those that don’t satisfy that property. The percentage of discarded distractors is the “power of disambiguation” of the agent. The reference agent will post the property in the form of an assertion (for example “the referent is green”, “the referent is close to a plant”, “the referent is on the other side of the room”...) and a set of distractors that satisfy that property.

A second agent can be applied to the output set of distractors from the first agent, trying to reduce the set even more. All the reference agents can be tested, looking for the set that has a higher power of disambiguation. Also, an evolutionary algorithm has been tested in our implementation. In this case, each individual is a set of reference agents. The fitness function adds the power of

each agent, and divides it by its position in the set, so that bigger sets have lower values. The operator exchanges agent subsets, and the mutation removes agents in a set or puts a new one in.

The goal of our design is to leverage the system’s ability to express itself using different combinations of the complete set of disambiguating properties made available in this manner. This gives system designers a choice between having many simple agents or fewer more expressive, complex agents. This choice should be considered in terms of particular implementation details.

It’s obvious that the disambiguation’s performance is better when the set of distractors is small. It is possible to obtain a better performance from the VG if the guide manager plans a set of actions that bring the user closer to the referent, and only tries to disambiguate when the user is close enough. The problem is where to place the limit between approximation and disambiguation: one can spend more time trying to get the user close to the referent, making the disambiguation almost trivial, or one can spend less time on the approach, and do a more complex disambiguation.

3.5 Alerts

The *alerts* subsystem is in charge of proactively looking after the user, and sending possible urgent alerts to the guide manager. If the warning information is more important than the guiding, the VG will have to delay instruction giving, and warn the user first. To decide about the importance of the warning part of the discourse, we defined *agents* as entities in charge of watching for special situations. Each agent takes care of a specific kind of situation that may imply some sort of hazardous or bad result. They are all independent, and may differ depending on the kind of environment, goals or even the kind of user.

Each agent has a weight that reflects its priority when being considered. An agent always evaluates its situation and returns a value in the $[0, 1]$ interval. A near zero value means there are low probabilities for the situation to happen and a near to one value means the situation is on the verge to happening. All agents that exceed a threshold value will be considered as contributors to the discourse. We sort them in descending order based on the result of multiplying each return value by the weight of the agent. If an agent is considered as a contributor, its warning is introduced in the

discourse.

We defined three types of agents: **information agents** watch for interesting hotspots in an area, **status agents** watch over the user's status, and **area agents** watch over special areas, including dangerous areas.

In our entry for the GIVE challenge there was a status agent that checked how much time had passed since the last user action to identify when the user might be lost. There was one agent that checked for booby traps the user might step on (some of them resulted in losing the game immediately). Another one ensured the user remained within a *security area* that abstracted all possible common routes to reach the intended destination. If a user leaves the security area, he is going in the wrong direction. This security area is dynamically updated attending to the current user's position. Finally, **alarm agents** watch for wrong actions, controlling if user is on the verge of pressing the wrong button or leaving the room using a wrong exit. We implemented no information agents, but they would be interesting in real situations.

4 Deficiencies

The biggest flaw in GUIDE is, without a doubt, the lack of feedback from the user and from the system. When the user presses H to ask for help, the system does nothing else than repeating the last sentence. This is not what one should expect. Usually the user does not need to read again the same sentence. What the user need is more information, or the same information expressed in a different way.

The GUIDE architecture can be modified to include a feedback module, and the rest of the modules can be altered to use this new information. For example, the guide manager could make use of families of guide agents to express the same instruction. Each guide agent in each family would be specially designed to express the same information in different ways. Also a specific level of the instruction tree could be forbidden when it's detected that the user does not like it (has failed to accomplish a task several times using that level).

The disambiguation module could change the importance of different reference agents, adjusting it to the number of times the user has failed to identify and object when using those agents. This way, efficient reference agents would be used more, while less efficient agents would be dis-

carded, and only used if there's no other way of referring an object. This could give a personalized guiding for each user.

Finally, the alerts subsystem could dynamically adjust the agent's weight, depending on how many times the user has triggered them.

This material has not been completely tested, but we believe it will be the base of a new research line, and that we will be able to add it to future editions of the challenge. It is proposed that the help information the user can ask for is expanded using text or voice recognition.

5 Examples

Figures 10, 11, 12, 13 and 14 present some representative examples of GUIDE. The comments for this figures explain what is happening, and the text in the screenshots shows the instruction.



Figure 10: Dissambiguating a button. Note the other green button outside the room. Disambiguation is logically needed here, but since actions never take place outside the actual room, disambiguation here seems irrelevant.

6 Issues

As constructive criticism, we will like to point at some aspects we think should be improved in future editions of the challenge.

Regarding the timing of the on-screen text. It would be useful to be able to explicitly specify the minimum amount of time that an instruction should stay on the screen, or at least have a (historic) stack of instructions, keeping the latest instructions on the screen using a gray font.

Regarding the list of visible objects. Many times a complex object, for example a plant, would logically block the sight of an object, while that



Figure 11: Warning the user, again. The user was warned before, but a reminder is always welcome.



Figure 14: Upsetting the system.



Figure 12: Another kind of alarm, this one tries to prevent the user from choosing the wrong exit.



Figure 13: Referring an object by all possible means, and noting the user is walking in the wrong direction.

object is obviously in sight, since it can be seen through the leaves. Considering that visual information is so important, these problems should be solved.

Regarding the graphical environment. Some textures and meshes are still confusing for the user. It was very bad in the first versions, where the camera angle together with wall colors were very confusing, and though it was mostly fixed, we believe it still needs improvement.

7 Enhancements

As we find this research line very interesting, we would like to give some advice we think will make future editions more attractive.

In our opinion, two aspects can be improved. The first thing is the set of classes in GIVE, and the second thing is about the complexity of virtual environments.

In relation to **the classes in GIVE**, it would be great to have a better and bigger framework, that would allow to connect different world representations (rather than only XML) or other graphic engines. A better access to the XML structure would let the developer add his own objects or group spaces according to his needs, being able to access this new information through the original API.

Knowing that these changes are not trivial, and require rethinking all the basics, we think that another way of improving the experience is **altering the complexity** of the maps. We would like to see maps that simulate urban environments: parks, roads, sidewalks, buildings with different levels. . . The increase in complexity would be better if the environment was less (or not at all) discretized.

8 Acknowledgments

This research is funded by the Ministerio de Investigación, Ciencia e Innovación (GALANTE: TIN2006-14433-C02-01), and Universidad Complutense de Madrid and Dirección General de Universidades e Investigación de la Comunidad de Madrid (MILU: CCG07-UCM/TIC 2803).

References

- D. Dionne, S. de la Puente, C. León, R. Hervás, and P. Gervás. 2009. A Model for Human Readable Instruction Generation Using Level-Based Discourse Planning and Dynamic Inference of Attributes Disambiguation. In *12th European Workshop on Natural Language Generation*, Athens, Greece, 03/2009.
- Alexander Koller, Johanna Moore, Barbara di Eugenio, James Lester, Laura Stoia, Donna Byron, Jon Oberlander, and Kristina Striegnitz. 2007. Shared task proposal: Instruction giving in virtual worlds. In Michael White and Robert Dale, editors, *Working group reports of the Workshop on Shared Tasks and Comparative Evaluation in Natural Language Generation*.
- E. Reiter and R. Dale. 1992. A fast algorithm for the generation of referring expressions. In *Proceedings of the 14th conference on Computational linguistics*, Nantes, France.