The GIVE-2 Challenge: Saarland NLG System

Johannes Braunias, Uwe Boltz, Markus Dräger, Boris Fersing, Olga Nikitina
Saarland University
Saarbrücken, Germany
{bjohanne, uboltz, mdraeger, borisf, nikitina}@coli.uni-saarland.de

Abstract

This paper presents the Saarland NLG System that was developed at Saarland University and participated in the Second GIVE Challenge. When guiding the instruction follower through the virtual environment, we adapt to changes in their visual field and use human-like referring expressions.

1 Introduction

This paper describes the Saarland NLG System that participated in the Second GIVE Challenge.

The task of this generation challenge is to implement a natural language generation system that guides the player (called the instruction follower) through a virtual 3D environment consisting of interconnected rooms (see Figure 1). The whole process is presented as a game, where the player follows the system's instructions to get a trophy from a hidden safe. In order to find and open the safe, the player needs to explore the virtual world and push a number of buttons that are distributed over the rooms of the game world. Buttons control the state of different objects in the GIVE world: they open and close doors, activate and deactivate alarm tiles, lock and unlock the safe[1].

All necessary actions that the player is supposed to perform in order to win the game are collected together in a plan. The task of our system is to translate the plan from its symbolic representation into a number of instructions in written English. The instructions are displayed on the screen, guiding the player to the right buttons in the appropriate order. In the ideal case, the player complies with these instructions, and, eventually, is able to pick up the trophy from the open safe.

[1]More information on the design of the GIVE framework and motivation behind it can be found in (Koller et al., 2010a).

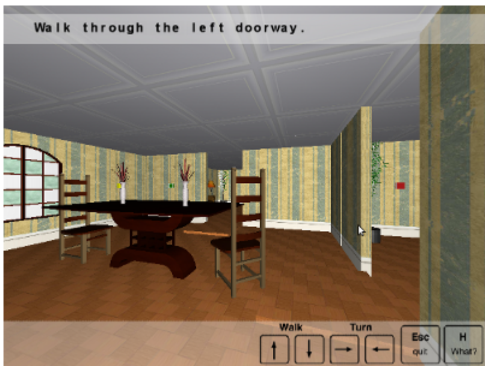

Figure 1: *A screen shot from a running game.*

Our main motivation was to find a more human-like apprroach to the task of instruction generation and to incorporate the information about changes in the visual field of the player into the instruction generation process. We observed that people, when they face the same task, tend to guide instruction followers to a place from which they can see the target of the next manipulation. Then, human instruction givers try to refer to the target using a referring expression. Our system explores this approach. Therefore, our navigation and referring expression generation strategies are tightly connected and support each other. The former limits the variety of targets that our system should be able to describe and the latter focuses on the efficient and human-like way to refer to them. Our control structure directs the overall process of the game, making our instructions sensitive to changes in the visual field of the player and adapting to their actions.

When designing our system, we focused on three main aspects:

- navigation strategy;
- referring expressions generation;
- high-level control structures.

Below, we describe them in details. First, Section 2 describes our approach to navigation. Section 3 explains the referring expression generation algorithm. Section 4 describes the control structures of our system, it shows how we decide when a new plan should be obtained, a new instruction should be sent to the player, when a player should be informed about possible consequences of his actions. Finally, Section 5 gives a summary of the project and compares our system with systems from the First GIVE Challenge.

2 Navigating in GIVE Worlds

2.1 Regions and the Plan

The GIVE Challenge focuses purely on the generation of instructions, and not on planning problems. Therefore it is not part of the challenge to decide what actions the player has to execute in order to win the game. That task is taken care of by the GIVE framework. It contains a planner that can be called at any time by the NLG system and that will provide a plan, a sequence of very simple actions the player has to perform. The task of the NLG system then is to translate that plan into natural language instructions.

Because the planner does not work for continuous worlds, GIVE worlds are divided into discrete regions. Any position in the world is in exactly one region. Regions are computed in such a way that every position in a region can be seen from every other position in the same region as well as every position in every region that is adjacent to it. Each region has a unique name by which it is referenced in the plan.

2.2 Navigation

The plan we get is too detailed to send its instructions to the player one by one. It contains one movement instruction for every region the player has to walk through. Quite often the player has to push a button that they can already see, but there are still several regions between them and the button. In these cases a single instruction, telling the player to push the button, would be enough. Sending one instruction per region, however, would mean first to make the player move several times and then to finally tell them to push the button. This kind of instruction giving would have a number of disadvantages:

- More instructions have to be created. This is no problem for the system, which can generate the necessary instructions fast enough, but the player must spend more time reading all the instructions.
- The player is on a very short leash. They do not have any freedom to choose their own route to their next target, as they have to walk through exactly the regions the planner selected.
- Referring to regions is usually harder than referring to concrete objects in the world. It is relatively easy to refer to the 'blue button to the right of the chair'. Regions on the other hand are very difficult to refer to, because they are invisible to the player. Where the player just sees a room, the system might see a lot of different regions. Referring to them would often require very complicated referring expressions, explaining where a particular region is refered to by spatial relations to walls and other objects.

Because of these disadvantages, our system tries to skip simple movement instructions whenever possible. It does so by identifying the important steps of the plan, that is, the actions the player has little or no way to avoid in order to win the game. These actions are pressing a button, taking the trophy and walking through a door.

In detail, instructions are created as follows:

The system finds the next step in the plan that requires the player to take the trophy or press a button. It then checks if the target object of this step is in the same room as the player or, if this is not the case, if the player can see the target object. If so, the system gives an instruction with a reference to this object. If the target object is neither visible, nor in the same room, that means that the player will have to walk through a door to get to it. So the system then looks for this door.

Unlike buttons and the trophy, doors are not mentioned in the plan. Therefore it requires some more effort to find the next door. The system checks in which room the player is currently located, then finds all doors this room has and gets the name of the regions affiliated with those doors. It then checks if, according to the plan, the player has to cross one of these regions. If so, that means the player is supposed to walk through that door, so the system makes this door its new target object.

Whether the target object turns out to be a door, a button or the trophy, the next step is always the same. First, the system checks if the target is visible. If so, the system tries to generate a referring expression for it (see Section 3). If a distinguishing referring expression can be created, an instruction with this referring expression is created. To this end, the referring expression is simply concatenated with another string. The choice of the string depends on the type of the target object. If the target is a button, the string will be 'Press' to get 'Press the red button to the right of the chair'. Doors are associated with string 'Walk through' and the trophy has the associated string 'Take'.

If no distinctive referring expression can be created, the system checks if the player can get closer to the target. If they can, there is a good chance that, once he does get closer, it will be easier to refer to the target object. So the system tries to make the player move closer to the target, simply telling them to "move forward". If the player cannot get any closer to the target, the system will generate an instruction with a referring expression that is not distinctive. The idea behind this is that when in doubt, the player will most likely try out the closest possible target first, so even if the referring expression is not distinctive, the chance is high that he will pick the right object.

If the next target object is not visible to the player, the system does not refer to it directly. Instead it first checks if the player could see the target if they would simply turn towards it, or if the sight line is blocked by a wall. If turning alone is enough, the system just creates a very simple instruction, telling the player to do so. To do this, it computes the angle between the orientation of the player vector and the vector that points from the player to the target object. This angle is then mapped to a list of English words that describe directions, and an instruction like "Turn right" or "Turn left a little" is generated.

If the player will not be able see the next target even if they turn, the line of sight is blocked by a wall. Therefore, the system tries to find a position on the way to the target which the player can see (cf. section 2.3) and tries to guide them there, again using simple "Turn" and "Walk forward" instructions.

2.3 Choosing the next Target Region

In order to create navigation instructions, we use a method to find the furthest possible region that is still visible to the player. As mentioned before, the GIVE environment is separated into rectangular regions. To find a position to guide the player to, the system checks whether the line of sight to the center of such a region is blocked by a wall or not, starting with the center of the next region the player has to walk through according to the plan. It then checks the center of the next region, and the next, and so on, until it either finds a region whose center cannot be seen from the player's position or until it gets to the original target, which it already knows is out of sight. The last center of a region that can still be seen then becomes the position to which the system will try to guide the player.

To find a position on the way to the next target which the player can see, the system starts by checking if the center of the next region the player has to walk through can be seen from the player's current position. It then checks the next region's center, and the next, and so on, until it either finds a region which center cannot be seen from the player's position or gets to the original target, which it already knows is out of sight. The last center of a region that can still be seen then becomes the position to which the system will try to guide the player.

Because the regions of the world are computed in a way that guarantees each position in a region can be seen from each position in all its adjacent regions, the system will always find at least one visible center of a region, and that is the one of the next region the player has to go through. So we can be sure that the player will always get an instruction that allows him to progress at least a little bit.

2.4 Checking Visibility

The GIVE framework provides a list of currently visible objects, i.e. objects that are displayed on the player's screen. To find out whether or not the center of a region or a target object can be seen from the player's position, however, this list is not is not sufficient. Consider Figure 2. It shows the player in a U-like room and a blue button, which is the target of his next action. The system can send two different navigation instructions: "Turn around" or "Walk to the right". To choose between them, we need to know whether the target will become visible after the player has turned.

In this case, knowing which objects are *currently* visible is not enough for creating clear and simple instructions. We also cannot rely on the

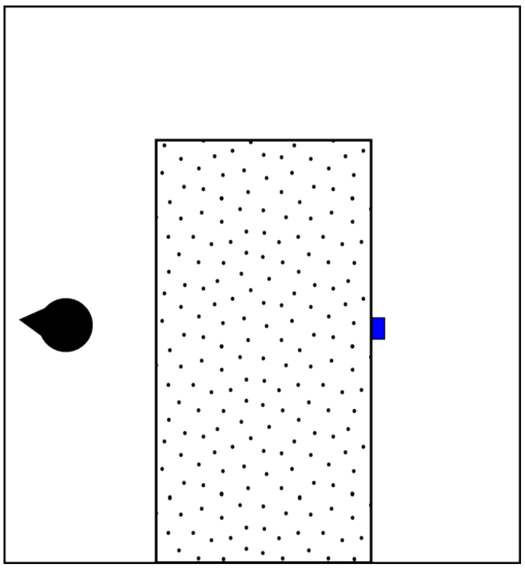

Figure 2: *A situation where the visibility check method might be useful.*

information about the position and orientation of both the player and the target. For example, if we only rely on this information, in the situation depicted in Figure 2 an instruction “Walk forward” would become correct after the player has turned around (and it is not).

To solve this problem we implemented a method which checks if a given a position is actually visible from another position.

The first thing the method does is to draw a line segment between the two positions. This line is the *visibility line*.

After the visibility line has been created, the method checks if it intersects with any wall or closed doors. If there is a closed door or wall between the two positions, the method returns that the target position is not visible from the source position.

Closed doors are a bit tricky because unlike walls, we might use them in a referring expression. The target of the referring expression for which we check the visibility always intersect with the visibility line. That means, that if we want to use a closed door in a referring expression, it will always intersect with the visibility line during the visibility check process and the method would report that the door is not visible. For this reason we had to create exceptions in the case where the target position corresponds to a closed door.

3 Referring expressions

This section describes the algorithm that governs the referring expression generation in Saarland NLG System. First, we explain why we have chosen to create a new algorithm instead of using one of the existing referring expression generation algorithms. Then we describe the algorithm in detail, with examples. We also explain why different strategies were chosen for different types of referents.

From the computational point of view, referring expressions are sets of attribute values that describe a referent (i.e. an object in the game world). For example, a referring expression “the red button” names values for two attributes: color (“red”) and type of the referent (“button”). The object that should be described by the referring expression is called the target object, and the objects from which the target object should be distinguished are called distractor objects.

We took an approach that is different from the known algorithms. Our motivation was to generate referring expressions that are more like those that human instruction givers produce in similar environment. To do that, we looked at the corpus of referring expressions collected in the human-human GIVE games (Koller et al., 2010b) and found a number of types of referring expressions that are used frequently. Each type of referring expressions had its own set of attributes that were used to describe the target object. For example, the simplest type of referring expressions only contained the type attribute (“the trophy”), another contained also the reference to the position of the target in a group of similar objects and a description of this group (“the right button in the group of two blue buttons”). The classical Incremental algorithm ((Dale and Reiter, 1995)), although it is psychologically motivated, still does not always generate the referring expressions with those attributes that human instruction givers include often. For example, humans tend to include the color of buttons into references even when there are no distractor objects of the same color. Therefore, we created our own algorithm.

We did not try to unveil the strategy that human instruction givers use (one of the reasons being that different people often prefer different strategies and come up with different referring expressions in similar situations). Instead, we created a simple search algorithm that works with lists of most often used types of referring expressions. Once the algorithm gets a set of distractors and a target object that it should single out, it goes over the list of the types of referring expressions and for each such type decides, whether it is applicable to the target and whether it is distinguishing given the distractors.

When we say of a referring expression of certain type that it is applicable to the target, we mean that the target has all the attributes mentioned in the referring expression of this type. For example, if we want to describe a referent as “the red button to the left of the flower”, there should be a flower to the left of the target.

When we say that a referring expression of certain type is distinguishing, we mean that it is true of the target object and false of all distractors. For example, “the red button to the left of the flower” will be a distinguishing description for a target that is a red button and that is situated to the left of a flower, and there is no such distractor that is also a red button and is situated to the left of the same (or of a different) flower, so that there is no other button that is situated between this flower and the distractor. We consider as distractors all objects that are in the field of view of the player at the moment when the referring expression generation algorithm is called. In other words, when determining whether a given referring expression is distinguishing, the algorithm only tries to single the target referent out from the set of those objects that are currently visible to the player.

Conditions under which a referring expression is applicable and distinguishing were modeled for each type of referring expressions separately. First, the algorithm checks if the referring expression is applicable, and if it is, it checks whether it is distinguishing. If it is not applicable, the algorithm goes on to the next type of referring expression. When the last type is checked and rejected, the algorithm fails. It is also possible to check applicability and distinguishing power of the referring expression, taking into account only distractors from the same region, or generate a referring expression that only names the target by its type and color, e.g. “the red button”.

The algorithm selects a referring expression based not only on its applicability and distinguishing power, but also on what is the type of the referent that is to be described. In other words, for different objects different lists of types of referring expressions are used.

We distinguish four different classes of referents: buttons, doorways, trophies and landmark objects. This distinction is based on our navigation strategy and on the GIVE-2 corpus study. Our navigation strategy is designed so that we only need to refer to buttons, doorways and the trophy. If we consider the GIVE-2 corpus, we will see that human instruction givers also refer to so called landmark objects (chairs, pictures, lamps, etc.) and use them to describe other targets, such as buttons.

Each class of referents is associated with its own list of referring expression types.

In the GIVE-2 Challenge each game had only one trophy. Thus, we can refer to it in a simple way, using a referring expression “the trophy”. Therefore, the list associated with the trophy contains only one type of referring expressions: naming by type.

For the landmark objects, we also have only one type of referring expressions: naming them by their type (“the chair”), which was also the way human instruction givers preferred to refer to them. Such behaviour might have been caused by the simple structure of the worlds, for which the GIVE-2 corpus was collected: these worlds did not contain similar landmark objects in one room, so there was no need to refer to the landmark objects in a more complex way. Despite that, our decision was to generate a referring expression without a landmark object or to use navigation instructions to guide the player closer to the target instead of using a longer referring expression that contains a description of a landmark object. For instance, we use a combination of instructions “Walk forward.”, “Press the red button to the left of the chair.” instead of “Press the red button to the left of the chair to the left of the lamp”.

The types of referring expressions that the algorithm generates for buttons and for doors are different. They are given below:

Types of referring expressions that are used for buttons.

- type and color of the referent (“the red button”)
- type and color of the referent and its position with respect to a landmark + the type of the landmark (“the red button to the left of the chair”)
- type and color of the referent and its position

with respect to a landmark + the color of the landmark ("the red button to the left of the green one")

- type of the referent and its position in a group of similar objects + the type and number of the objects in the group ("the right button in the group of 3 buttons")
- type of the referent and its position in a group of similar objects + the type, color and number of the objects in the group ("the right button in a group of 3 red buttons")
- type and color of the referent and its position in a group of the similar objects ("the closest red button")

Types of referring expressions that are used for doorways.

- type of the referent ("the doorway")
- type of the referent and its position with respect to a landmark + the type of the landmark ("the doorway to the left of the chair")
- type of the referent and its position in a group of similar objects ("the right doorway")
- type of the referent and its position in the visual field of the player ("the doorway on your left")

For buttons, the color attribute is always included, since the majority of human instruction givers do so. Also, buttons can be distinguished by color, which makes it possible to use one button as a landmark in the description of another button ("the red button to the left of the green one"). Buttons are often situated in groups. That makes it possible to use references to groups of buttons as part of the target description. Group references not only exclude many possible distractors, but also make it possible to use spatial adjectives in references, e.g. "right". Otherwise, these adjectives are computationally hard to model. Finally, we also used the distance to the player as one of the attributes for buttons ("the closest red button"). This type of referring expressions was not often used by human instruction givers, but such references turned out to be very effective in some situations, when all other strategies failed.

As for the doors, there were only two types of referring expressions that human instruction givers used quite often. First, they named the doorway by its type; second, they also described them with a spatial adjective if there was more than one door in the field of view of the instruction follower. We have added references to doorways using a landmark object ("the doorway to the left of the chair") and references that describe doorways with respect to its position in the visual field of the player ("the doorway on your left"). We needed more different referring expressions for doors because of our navigation strategy. When guiding the player, our system often refers to doorways, sometimes from far apart. The further the player is from the target, the more distractors they see, and references with spatial adjectives might not be feasible.

The lists of the referring expression types are ordered. The bigger the cognitive effort for the player to understand the referring expression, the later it comes in the list. In this, our algorithm follows the idea presented in (Kelleher and Kruijff, 2006). First, we aim for the simplest possible referring expression (naming the target by its type or by its type and color only, e.g "the doorway" or "the red button"). In case this referring expression is not distinguishing, references with landmarks are used. We prefer larger and therefore more salient landmarks (chairs, flowers, etc.) to smaller ones (buttons). We also believe that shorter descriptions should be preferred to longer ones, and thus in our algorithm we placed the referring expressions with groups after the referring expressions to landmarks. For the same reason, we prefer referring expressions with spatial adjectives to references that describe the referent by its position in the visual field of the player, i.e. "the right doorway" to "the doorway on your right". Finally, the referring expressions that use the distance to the player ("the closest red button") were placed in the very end, because we considered them to be useful in a smaller number of situations than all other descriptions.

3.1 References to groups and spatial adjectives

An important feature of the algorithm is that it takes the mutual disposition of objects into account. In other words, we can generate group references and we can use spatial adjectives to refer to objects.

All objects in the GIVE environment have three coordinates (x, y, z). We define the notion of group as a set of objects, that:

- are of the same type;
- have all the same value of x, y or z coordinate;
- for each member of the group, its other two coordinates differ from the corresponding coordinates of its nearest neighbour not more than some value d. We set the value d to a certain constant, but it might be more precise to condition this value on the size of the objects that form a group.

For example, three buttons with coordinates (1, 2, 3), (1, 2.2, 2.7) and (1, 1.9, 2.9) will form a group, if d is set to 0.3, a button with coordinates (1, 2, 4) will be out of the group (see Figure 3). On the other hand, three buttons with coordinates (1, 2, 3), (2, 3, 1) and (3, 2, 1) will not form a group, if d is set to a value that is smaller than 1.

To find groups of buttons given the target, we first find a similar object that is close enough to the target to form the group. When we find such object, we remove it from the set of distractors. Then we look in the set of remaining distractors for such objects that are close to one of the objects from the group.

The use of group references helps to distinguish the target from all distractors that do not belong to the group. To disambiguate between buttons inside the group, we use spatial adjectives. The meaning of these adjectives is such that if it is true of one object, it cannot be true of another object from the same group.

We model the meaning of 6 spatial adjectives: “top”, “bottom”, “left”, “right”, “middle”, “center”. To find the top and the bottom objects, we compared them by their coordinate on the vertical axis. To find the right and the left objects, we compare the angles between them and the two-dimensional vector that goes from the player’s position towards the objects’ positions (see Figure 3). The objects with the highest and lowest angles were the objects from the group periphery. Using some additional information and methods from the GIVE Java packages, we were able to find out which of them is the right one and which of them is the left one. On Figure 3 we can see that angles alpha and beta are the smallest and the greatest angles. Thus, we know that buttons b3 and b1 are the periphery buttons.

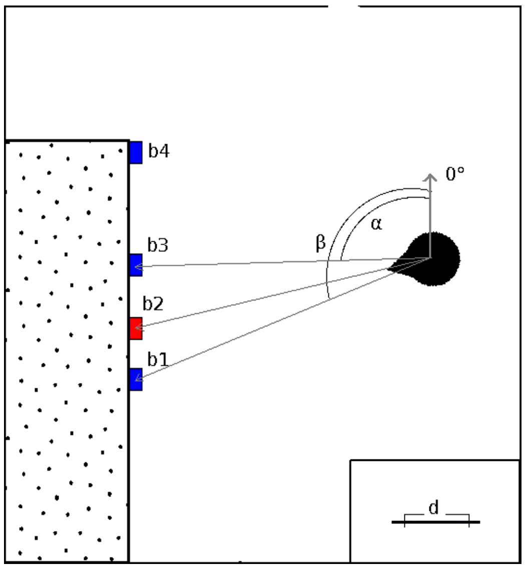

Figure 3: *The detection of a group of buttons.*

The notions of the middle and center objects were defined for groups where objects were arranged approximately along parallel lines. In addition, these groups should have three objects that are arranged along one line at maximum. In such groups, the middle object is the object that is neither left nor right in a row or in a column. If a group had more than one row, we added to the description both its vertical and horizontal positions, for instance, “the right middle button” or “the top middle button”.The group from Figure 3 includes 3 buttons, and we know that the button b2 is not a button from the periphery. So, it is the middle button.

The center object was defined only for a configuration with evenly distributed 9 objects that were situated close to each other. In such group, the center object would be the object that is neither top or bottom, nor left or right.

In practice, group references were used only with respect to button targets. Methods for calculating right, left and middle objects were also used to construct referring expressions for doorways.

3.2 Advantages and disadvantages of our algorithm

Our algorithm is able to cope with many situations due to a rich and carefully chosen variety of attribute sets. It also generates referring expressions similar to those that human instruction givers tend to produce. The separation of conditions that determine if the referring expression is applicable and whether it is distinguishing enables the developers to modify the algorithm locally, adding, editing or removing only certain types of referring expressions. The algorithm is tightly connected with our navigation strategy; it helps to decrease the number of instructions that are sent to the player and to avoid making the navigation instructions precise by explaining to the player what distance they should cover and at which place they should stop.

On the other hand, the algorithm is very domain-specific. Also, the algorithm is only as good as the modules that model each particular type of referring expressions. In particular, due to implementation imperfections, in some situations ambiguous referring expressions are considered to be distinguishing, which might confuse the player.

4 Reacting to what the player does

4.1 Alerts

To make our instructions well-timed, we needed a mechanism to decide when the next instruction should be sent. To achieve this functionality, we implemented an *alert* mechanism.

An alert is an object that consists of a string method that creates a message to the player, and a boolean method which determines when the message should be sent. The system constantly calls these methods of all active Alert objects, and when one of them returns true, the system creates the message of this alert and sends it to the player. The alerts are created when an instruction is created, covering events that might happen after that instruction is sent.

The alerts work very well when it comes to controlling when a new instruction is sent. When the player moves, the last instruction might still be displayed on the screen and become invalid. If the instruction says "Turn left" and the player turns to far, he now has to turn right. The system has to realize that and send a new instruction, otherwise the player might turn even further in the wrong direction. This is prevented in the following way: When, for example, the instruction "Turn left" is sent, an alert object is generated that will be activated when the player faces in the right direction. At that moment it will trigger the creation of the next instruction.

Alerts are a very powerful tool, because they might be used to give feedback to the player. When, for example, an instruction to walk through a door is created, the system could create an alert containing the message "No, that was the wrong door" and a method that returns true if the player has walked through the wrong door. Furthermore, when an instruction to press a button is created, the system could create an alert containing the message "No, it's in the other direction" and a method that checks at which angle to the target button the player moves next. This way the player will be told when he does something wrong. In many cases, one can think of this as sticking post-its to objects in the world, telling the player "this is the right one" or "this is wrong, turn around". However, in our system this was not implemented due to the lack of time, although this was the original idea.

An advantage of the concept of alerts is that a programmer who creates an instruction for a certain situation only has to think about that particular situation when he sets up alerts for it. He can completely ignore other cases, which makes it easy to separate the task of creating instructions between multiple programmers. The same is true for different types of alerts: To create a new type of alert one does not need to know anything about how other types work. All it takes is creating one method that, in whatever way, returns a string, and one that returns a boolean.

4.2 The alarm tile warning

An important feature we wanted to implement is the alarm tile warning. Some parts of the floor in the world are alarms. If the player steps onto an active alarm tile, they lose the game. Thus, we deemed it important to find a way to tell the player to avoid active alarm tiles.

One thing we do in our system is to explicitly tell the player to avoid the red parts of the floor already in the very beginning of the game.

The second thing we decided to do was to implement a method that triggers a warning when:

- the player enters a room which contains active alarm tiles
- the player sees an alarm tile and is near to it

This alarm warning system is based on alerts which are set to be triggered as soon as possible.

For the first condition, the method generates an alert with the message "Beware of the alarm you see!" and for the second condition it generates the alert with the message “Beware of the alarm in this room!”.

For the second condition, the method also checks if the distance between the alarm tile and the player is below a given threshold, as the alarm warning should not annoy the player when the alarm is too far away. In order to determine this threshold, we also considered the speed of the game on different computers, the warning should not be send too late on slow machines.

4.3 When to replan

Our system does not need to decide what actions the player has to do next, the planner provided by the GIVE framework solves this task. However, our system has to decide when to call the planner. Computing a plan takes several seconds, so recomputing after every given instruction is not an option, because it would delay the next instruction too much.

Our approach is therefore to replan as rarely as possible. The system only calls the planner if the current plan becomes invalid. This can happen for two reasons. Either the player pressed a wrong button, which might make it necessary to press additional buttons to reverse the wrong button's effects. Or the player moves to a wrong place. In that case it might happen that they are so far away from any region the plan tells them to move to, that it is beyond our system's capabilities to guide them there.

So we replan only if the player presses a wrong button or gets too far away from the way they are supposed to walk. The latter we find out by computing the distance between the player's current position and every region that they have to walk through according to the plan. If the smallest distance is below a certain threshold, that means that even the nearest region is too far away and that replanning is necessary.

5 Conclusion and related work

In the first edition of the GIVE challenge, the participating systems introduced a number of interesting ideas on how to improve instruction generation. Our system shares some of those features, while striving to add further improvement.

One common feature to all GIVE systems is to filter or aggregate to some extent the output generated by the planning module. Due to the discretized world structure in GIVE1, most aggregation amounted to summing up subsequent movement instructions into one (e.g. ”move forward five steps” instead of ”move forward one step” for five times).

In GIVE2, however, the game worlds were discretized into larger regions of different shape and size, which were not as easy to handle. Instead, we implemented a visibility-based aggregation routine that is both able to sum up several movement instructions and join instructions of different kinds (e.g. ”Walk left and push the green button”).

Another aspect that we were able to improve upon was the use of *alerts*. Alerts were introduced in GIVE1 by the Madrid system (under the name of *alarms* (Dionne et al., 2009)) to recognize undesired player behavior (heading in the wrong direction, or walking towards an alarm tile). In the Saarland NLG system, we extended the functionality of alerts: Due to the merely coarsely discretized worlds of the GIVE2 challenge, it can be quite difficult to track movements of the player and coordinate them with the output of new or updated instructions. In particular, instructions displayed on the screen might become invalid during player movements. In our system, we use alerts to achieve a better timing of instruction giving.

An aspect that in our opinion has not been paid enough attention to in GIVE1 is the generation of referring expressions. As our system relies heavily on landmark objects in the generation of instructions, we felt that improvement on this aspect was essential. While many GIVE1 systems used off-the-shelf generation algorithms, we implemented our own, based on linguistic evidence from a corpus of transcribed GIVE game runs (see (Koller et al., 2010b)). Thus, our RE generation module imitates the behavior of an actual human instruction giver in the GIVE game context.

6 Acknowledgments

We are very grateful to Alexander Koller for his help, inspiration and constructive criticism throughout this project and to Konstantina Garoufi for her attention, valuable remarks and fruitful discussion that helped us to write this paper. We would also like to thank those who tested early versions of our system for their feedback and suggestions.

References

Robert Dale and Ehud Reiter. 1995. Computational Interpretations of the Gricean Maxims in the Generation of Referring Expressions. *CoRR*, cmp-lg/9504020.

Daniel Dionne, Salvador de la Puente, Carlos León, Raquel Hervás, and Pablo Gervás. 2009. Guide.

John Kelleher and Geert-Jan Kruijff. 2006. Incremental generation of spatial referring expressions in situated dialogue. In *In Proceedings of Coling-ACL '06, Sydney Australia*.

Alexander Koller, Kristina Striegnitz, Donna Byron, Justine Cassell, Robert Dale, Johanna Moore, and Jon Oberlander. 2010a. The first challenge on generating instructions in virtual environments. Emiel Krahmer and Mariet Theune (eds.), "Empirical Methods in Natural Language Generation". Springer.

Alexander Koller, Kristina Striegnitz, Andrew Gargett, Donna Byron, Justine Cassell, Robert Dale, Johanna Moore, and Jon Oberlander. 2010b. Report on the second nlg challenge on generating instructions in virtual environments (give-2). In *Proceedings of the International Natural Language Generation Conference (INLG)*, Dublin.