Generating Instructions at Different Levels of Abstraction

Arne Köhn[*] and **Julia Wichlacz**[*] and **Álvaro Torralba**[**] and
Daniel Höller[*] and **Jörg Hoffmann**[*] and **Alexander Koller**[*]

[*]Saarland Informatics Campus
Saarland University
{koehn, koller}@coli.uni-saarland.de
{wichlacz, hoeller, hoffmann}@cs.uni-saarland.de

[**]Department of Computer Science
Aalborg University
alto@cs.aau.dk

Abstract

When generating technical instructions, it is often convenient to describe complex objects in the world at different levels of abstraction. A novice user might need an object explained piece by piece, while for an expert, talking about the complex object (e. g. a wall or railing) directly may be more succinct and efficient. We show how to generate building instructions at different levels of abstraction in Minecraft. We introduce the use of hierarchical planning to this end, a method from AI planning which can capture the structure of complex objects neatly. A crowdsourcing evaluation shows that the choice of abstraction level matters to users, and that an abstraction strategy which balances low-level and high-level object descriptions compares favorably to ones which don't.

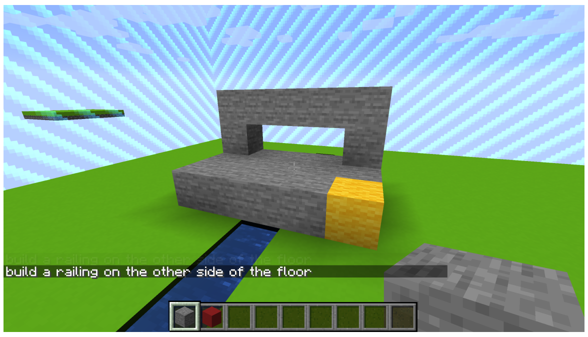
1 Introduction

Technical instructions in complex environments can often be stated at different levels of abstraction. For instance, a natural language generation (NLG) system for tech support might instruct a human instruction follower (IF) to either plug "the broadband cable into the broadband filter" or "the thin white cable with grey ends into the small white box" (Janarthanam and Lemon, 2010). Depending on how much the IF knows about the domain, the first, high-level instruction may be difficult to understand, or the second, detailed instruction may be imprecise and annoyingly verbose. An effective instruction generation system will thus adapt the level of abstraction to the user.

In this paper, we investigate the generation of instructions at different levels of abstraction in the context of the computer game "Minecraft". Minecraft offers a virtual 3D environment in which the player can mine materials, craft items, and construct complex objects, such as buildings and machines. The user constructs these objects bit by bit from atomic blocks, and it is always possible to generate natural-language instructions which describe the placement of each individual block. However, it can be more effective to generate more high-level instructions. In the bridge-building example shown in Fig. 1, it is probably better to simply say "build a railing on the other side" instead of explaining where to place the seven individual blocks – provided the IF knows what a railing looks like. Minecraft is the best-selling video game of all time (200 million users), which means that there is a large pool of potential experimental subjects for evaluating NLG systems. Minecraft has been used previously as a platform for experimentation in AI and in particular for NLG (Narayan-Chen et al., 2019; Köhn and Koller, 2019).

We present an instruction giving (IG) system which guides the user in constructing complex objects in Minecraft, such as houses and bridges. The system consists of two parts: a hierarchical planning system based on *Hierarchical Task Networks (HTN)* (Ghallab et al., 2004; Bercher et al., 2019), which computes a structured instruction plan for how to to explain the construction; and a chart-based generation system which inputs individual plan steps and generates the actual instruction sentences (Köhn and Koller, 2019).

Planning systems generate plans based on expressive declarative models, making this approach easily applicable to a wide range of domains, and giving it the power to deal with large degrees of freedom



Instructing the railing (here already built):

(a) "Put a block on top of the blue block. Put a block on top of the previous block. [. . .]"

(b) "Now I will teach you how to build a railing. Put a block on top of the blue block. [. . .]"

(c) "Build a railing from the top of the blue block to the back right corner of the floor."

Instructing a second railing to be built:

(d) "Put a block on top of the yellow block. Put a block on top of the previous block. [. . .]"

(e) "Build a railing on the other side of the floor."

Figure 1: Different possibilities to instruct the IF to build a part of a bridge.

in instruction generation. In particular, we leverage the hierarchical planning system to obtain three different strategies for describing complex objects as illustrated in Fig. 1: *low-level*, always instructing block-by-block (sentences (a) and (d)); *high-level*, always instructing to build the next complex subobject (sentences (c) and (e)); and a *teaching* strategy which first explains how to construct a complex subobject, and then uses high-level descriptions for that object in subsequent instructions (sentences (b) and (e)). We realize these strategies through designing the planner's action-cost function. The strategies constitute a first step towards a fully generic IG system which adapts its description of complex objects to the user's knowledge. The planner's cost function could also be used to incorporate additional criteria, e. g. that the generated sentences should be easy to understand in context.

We evaluate our IG system by crowdsourcing, using the open-source MC-Saar-Instruct platform for IG experiments in the Minecraft domain (Köhn et al., 2020). Each user receives instructions for building either a house or a bridge using one of the three abstraction strategies sketched above. The results show significant differences in completion times and user satisfaction across the three strategies. In the bridge scenario, the teaching strategy outperforms the high-level strategy in completion time and the low-level strategy in user satisfaction. In the house scenario, the high-level strategy outperforms the others in completion time for the walls, illustrating the importance of modeling user knowledge when choosing the level of abstraction.

Plan of the paper. After reviewing some related work (Section 2), we overview the architecture of our IG system (Section 3). We briefly introduce HTN planning, and how we use it to define hierarchical Minecraft construction planning models (Section 4). We then explain how to get from *construction* planning models to *instruction* planning models, and how to generate instruction plans at different levels of abstraction (Section 5). Section 6 describes our evaluation; Section 7 concludes.[1]

2 Related Work

Generating natural-language instructions grounded in the mechanics of a real or virtual world is a well-established type of NLG task. For instance, the GIVE Challenge (Koller et al., 2010) required NLG systems to guide human users through a maze while referring to locations and objects. We follow the GIVE Challenge in situating the task in a virtual environment and using crowdsourced task-based evaluation, but the Minecraft world is much more complex than the GIVE world, and contains complex objects.

Rookhuiszen et al. (2009) describe an IG system for the GIVE Challenge which dynamically adapts the level of detail of navigation instructions. They use a simple heuristic to switch between abstraction levels. Janarthanam and Lemon (2010) generate referring expressions at different levels of abstraction in an electronics repair scenario, and adapt to novice vs. expert users. They assume a finite set of possible descriptions for each object, and do not exploit the internal structure of complex objects.

The use of AI planning for elaborating and organizing the things that need to be said in an NLG system (i. e. *discourse planning*) dates back into the early days of NLG (Appelt, 1985; Hovy, 1988). Garoufi and Koller (2014) use non-hierarchical planning to compute communicative plans. They use planning

[1]Software and data are available at `http://redacted`

operators which encode communicative actions, and allow them to have effects both on the communicative state and on the state of the world; we disentangle these effects in our hierarchical planning model. Behnke et al. (2020) use a hierarchical planner to generate technical instructions at different levels of abstraction, but their system can only utter sentences which were stored with the planning operators as canned text. Their evaluation does not show that users prefer their system over a baseline, illustrating the difficulty of generating instructions at the right abstraction level.

The Minecraft domain has been used extensively for various tasks in AI (Aluru et al., 2015; Parashar et al., 2017), including planning (Roberts et al., 2017) and natural language understanding (Gray et al., 2019). Regarding NLG specifically, Narayan-Chen et al. (2019) trained a neural model to generate building instructions in Minecraft; in the absence of symbolic domain knowledge, their model struggles to generate correct instructions. Köhn and Koller (2019) show how individual instructions can be generated in Minecraft. Their focus is on generating indefinite referring expressions to objects which do not exist yet because the user is supposed to build them. Here we do not address how to generate the individual utterances, but how to determine the semantic content of these utterances.

3 NLG System Architecture

Our overall IG system consists of two separate modules: an *instruction planner* and a *sentence generator*. The role of the instruction planner is to compute an *instruction plan*, i. e. a sequence of *instruction actions*. An instruction action is an abstract semantic representation of a sentence; the sentence generator then translates it into a natural-language utterance, such as "place a block on top of the yellow block", "build a floor from the black block to the yellow block", or "now I will teach you how to build a railing".

The technical focus of this paper is on the instruction planner. Given a state of the Minecraft world and a specification for the complex object the IF is supposed to build (e. g. a bridge or a house), the instruction planner will compute a sequence of instruction actions as explained above. One technical contribution of this paper is that the instruction actions can be at different levels of abstraction; for example, the instruction plan can simply say "build a railing on the other side" in the situation of Fig. 1, or it can explain how to build the railing block by block. To ensure the correctness of the instruction plan, i. e. to ensure that the instructions, followed correctly, actually do result in the intended complex object, the instruction planner performs *construction* planning as part of its planning process: it internally refines its high-level actions into block-by-block plans and checks that these work.

The sentence generator takes instruction actions as input and produces natural-language utterances. We use the chart generation system of Köhn and Koller (2019) to generate sentences. This system generates sentences while simultaneously generating definite and indefinite referring expressions (REs); definite REs are used to refer to objects which already exist in the Minecraft environment, and indefinites are generated to refer to objects which do not exist yet because the IF is supposed to build them. Thus for instance, the sentence generator might translate the instruction action `ins-railing`(0,1,4,5,south) to "build a railing from the top of the blue block to the top of the red block" or "build a railing on the other side of the floor", depending on the state of the dialogue and the Minecraft world.

4 Hierarchical Construction Planning

As indicated, we distinguish between *construction* planning, which determines how a complex object can be built block-by-block; and *instruction* planning, which determines how, and in particular at which level of abstraction, the building of a complex object can be explained to the IF. Instruction planning encompasses construction planning as a sub-problem. We employ *hierarchical* planning for both tasks, with hierarchies of complex subobjects and the associated building activities. These are not required for construction planning per se (which can always proceed on a block-by-block basis). But they speed up the construction planning process, and they are key to instruction planning as proposed here. Previous work on planning in Minecraft (Roberts et al., 2017) has considered construction planning only, and has not considered hierarchical planning. We now introduce our construction planning models, which we will extend to instruction planning models in Section 5.

4.1 Background: HTN Planning

Hierarchical Task Network (HTN) planning (see Bercher et al. (2019) for a recent overview) comes with different levels of abstraction regarding the things that must be done, the *tasks*. *Primitive tasks* (also *actions*) can directly be executed in the environment. They come with conditions that need to hold to make them applicable, and their application changes the environment. *Abstract tasks* describe behavior at a higher level of abstraction. They are not applicable directly, but must instead be divided into other tasks by using *decomposition methods*. The new tasks may, again, be abstract or primitive. Methods are similar to derivation rules in a formal grammar where the left-hand side the abstract task and the right-hand side is its decomposition into other tasks/actions. Here we use *totally-ordered* HTN planning, a common subclass where the right-hand side is restricted to be a task sequence. Planners are given the overall tasks to accomplish, e. g. *build a bridge*. These are decomposed until only actions are left, which must be applicable in the initial state of the system.

We now give a definition using the basic formalism introduced by (Behnke et al., 2018). An HTN planning problem $\mathcal{P}$ is a tuple (F, C, A, M, tn_I, s_0):

- F is a set of propositional state features used to describe the environment. A *state* s is a truth assignment to these features, usually represented by the set of features true in the state.
- C and A are sets of abstract (also compound) tasks and primitive tasks (also actions).
- $M \subseteq C \times (C \cup A)^*$ is the set of decomposition methods (where * is the Kleene operator).
- $tn_I = (C \cup A)^*$ is the initial task network, $s_0 \in 2^F$ is the initial state of the environment.

Furthermore $\mathcal{P}$ is associated with functions $prec$, add, and del that map each action to its preconditions, add-effects, and delete-effects. $prec$ is a logical formula over F such that an action a is applicable if $prec$ is satisfied in current state s. When a is applicable in s, the state resulting from its application is defined as $(s \setminus del(a)) \cup add(a)$ where $add, del \subseteq F$. The sets of all possible states and actions (implicitly) define a state transition system describing how the environment can change.

Plans in HTN are defined through *task networks*, which are sequences in $(C \cup A)^*$. If $tn = \omega c \omega''$ with $c \in C$ is a task network and $m = (c, \omega') \in M$ is a method, then tn can be decomposed with m into $tn' = \omega \omega' \omega''$. A plan tn_S is a sequence in A^* that can be obtained by iteratively decomposing the initial task network, and that is applicable in the initial state. Plan quality is measured in terms of a cost function, $cost : 2^F \times A \to \mathbb{R}_0^+$, where the task of a planner is to minimize the summed up cost of the resulting plan.

Deciding plan existence in the framework we use here is, in general, EXPTIME-complete (Erol et al., 1994; Alford et al., 2015). A range of solvers is available that tackles this complexity through search (Nau et al., 2003; Bercher et al., 2017; Höller et al., 2020) or compilation into simpler frameworks (Alford et al., 2009; Behnke et al., 2019a). Some solvers provide guarantees on the solution costs (Behnke et al., 2019b), or enable anytime behavior continuing search to find better plans (Wichlacz et al., 2020).

4.2 HTN Construction Planning Models for Minecraft

As a compact explanation of our HTN construction planning models, consider the part of Fig. 3 that is highlighted in boldface, which shows a construction model for building a bridge. The propositional state features F take the form *block(x,y,z)*, encoding whether or not there is a block at those coordinates. The construction of any complex object in Minecraft can ultimately be decomposed into `put-block(x,y,z)` actions, which correspond to the primitive tasks in the HTN model.[2]

While all our Minecraft construction models share the same state features and primitive actions, they differ on the complex objects that can be built, $\mathcal{O}$. In the case of a bridge, these are the floor, the railings, and rows of blocks, placed at different positions and in different orientations. Each complex object X has an associated abstract task BUILD-X. For example, the task BUILD-BRIDGE(0,0,0,5,3,NORTH) corresponds to building a bridge of size 5x3 facing north and starting at position 0,0,0. A construction model specifies one or more decomposition methods for each abstract task, corresponding to different ways to build the object. This is illustrated in Fig. 3 by the two decomposition methods for BUILD-RAILING. The HTN planning system, run on such a model, will choose which option is more suitable for the task at

[2]In general, every `put-block(x,y,z)` action would need to be guarded with a precondition that a block can be placed at (x,y,z), i. e. that it is not placed in thin air). In our case, such preconditions are not necessary since the task hierarchy ensures this.

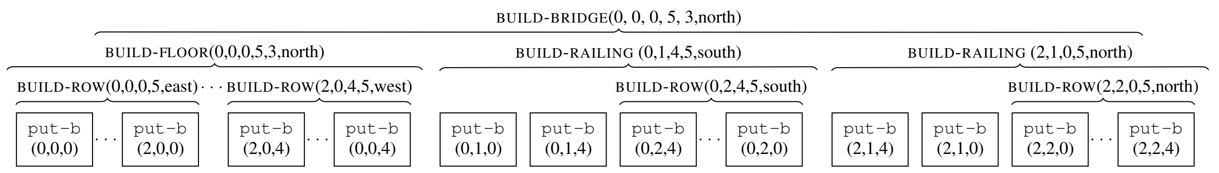


Figure 2: Example of an HTN plan for building a bridge of width 3, and length 5 at position (0, 0, 0).

hand to minimize the overall action cost. In Section 5.2, we will specify cost functions suited to optimize instructions for an IF.

Fig. 2 shows a plan. BUILD-BRIDGE(0,0,0,5,3,north) is decomposed into a floor and two railings. Other decompositions are possible, e. g., constructing the railings in different order or direction. The tasks are further decomposed until a valid sequence of `put-block` actions is reached.

Minecraft construction models – in general, and in particular the models we devise here – are challenging for HTN planning systems due to the large number of objects required to characterize a 3D world. In our experiments, we use recent algorithms based on Monte-Carlo Tree Search, which do not require a grounding pre-process and are thus able to scale to comparatively large Minecraft worlds while optimizing plan cost (Wichlacz et al., 2020).

5 From Construction Planning to Instruction Planning

Having discussed how to compute a *construction* plan with an HTN planner, we will now explain how to compute *instruction* plans. The actions in an instruction plan represent communicative actions, in which the IG system sends a sentence to the IF. These actions can describe the intended construction steps at different levels of abstraction. We capture this in the HTN model by defining a task for instructing the IF to build each complex subobject, say a railing. The planner can then choose to either achieve this task with a single primitive instruction action (which might send "build a railing" to the IF), or to further decompose it into smaller tasks, which will instruct the IF to build two blocks and connect them with a row.

One key challenge is that while the instruction actions represent natural-language instructions, they still need to be grounded in activities in the Minecraft world: we must guarantee that, if the IF follows the instructions correctly, then the correct complex object results. To this end, we reason about the construction and its explanation simultaneously. For every instruction action, the plan also contains the corresponding block-by-block construction actions, thus validating the instruction plan.

5.1 HTN Instruction Planning Model

We extend the construction model of Section 4.2 to an instruction model by adding the non-boldface parts of Fig. 3. Since we plan for instruction giving, we have to consider the IF and their knowledge. Some complex objects might be known to the IF (e. g. what a row of blocks is), others might not be. For example, if we instruct the IF to build the first railing for the bridge depicted in Fig. 1, the IF most likely does not know the exact shape this railing should have. Thus, we introduce a propositional state feature *knows-T* (see F in Fig. 3) for each kind of complex object T, representing whether or not the IF knows how to build such objects. We also incorporate information about the block the IF was instructed to place last. This is used by the cost function (see Section 5.2) to take into account that referring expressions are easier to generate, and to understand, for positions adjacent to the last placed block.

The instruction model also has new primitive actions A, whose names start with `ins-`; executing such an action corresponds to generating a sentence and sending it to the IF. First, for every instance X of a complex subobject, and for every possible block X, there is an action `ins-X` that represents asking the IF to build X. For complex subobjects, this action has the precondition that the IF knows the corresponding high-level concept. Second, the actions `ins-teach-start-X` and `ins-teach-end-X` correspond to utterances like "I will now teach you how to build a railing". These teaching actions have no preconditions and add `knows-X` to the state.

We furthermore add compound tasks INS-BUILD-X into C, which coordinate the instruction for object X, through different decomposition methods in M. These methods play a crucial role in our approach, as

$$\begin{aligned}
\mathcal{O} = \{ &\mathrm{bridge(x, y, z, length, width, orientation), floor(x, y, z, length, width, orientation),} \\
&\mathrm{railing(x, y, z, length, orientation), row(x, y, z, length, orientation)} \\
&\mathrm{with\ x, y, z, length, width} \in \mathbb{N}, \mathrm{orientation} \in \{\mathrm{north, south, east, west}\}\} \\
F = \ &\{\boldsymbol{block(x,y,z)}, lastblock(x, y, z) \mid x, y, z \in \mathbb{N}\} \cup \{\textit{knows-row, knows-floor, knows-railing}\} \\
C = \ &\{\textbf{BUILD-}\boldsymbol{X} \mid \boldsymbol{X} \in \mathcal{O}\} \cup \{\text{INS-BUILD-BLOCK}(x, y, z) \mid x, y, z \in \mathbb{N}\} \cup \{\text{INS-BUILD-X} \mid X \in \mathcal{O}\} \\
A = \ &\{\mathtt{\textbf{put-block}}\textbf{(x,y,z)}, \mathtt{ins\text{-}block}(x, y, z) \mid x, y, z \in \mathbb{N}\} \cup \\
&\{\mathtt{ins\text{-}X}, \mathtt{ins\text{-}teach\text{-}start\text{-}X}, \mathtt{ins\text{-}teach\text{-}end\text{-}X} \mid X \in \mathcal{O} \setminus \{\mathrm{bridge(x, y, z, l, w, o)}\}\} \\
tn_I = \ &\text{INS-BUILD-BRIDGE}(0, 0, 0, 5, 3, north)
\end{aligned}$$

$$\begin{aligned}
M = \{\textbf{BUILD-RAILING}\boldsymbol{(x,y,z,len,east)} &\to \langle \mathtt{put\text{-}block}(x, y, z), \mathtt{put\text{-}block}(x + len - 1, y, z), \\
&\quad \text{BUILD-ROW}(x + len - 1, y, z, len, west)\rangle, \\
\textbf{BUILD-RAILING}\boldsymbol{(x,y,z,len,east)} &\to \langle \mathtt{put\text{-}block}(x, y, z), \text{BUILD-ROW}(x, y, z, len, east), \\
&\quad \mathtt{put\text{-}block}(x + len - 1, y, z)\rangle, \\
&\dots \\
[L]\ \text{INS-BUILD-RAILING}(x, y, z, len, east) &\to \langle \text{INS-BUILD-BLOCK}(x, y, z), \text{INS-BUILD-BLOCK}(x + len - 1, y, z), \\
&\quad \text{INS-BUILD-ROW}(x + len - 1, y, z, len, west)\rangle, \\
[H]\ \text{INS-BUILD-RAILING}(x, y, z, len, east) &\to \langle \mathtt{ins\text{-}railing}(x, y, z, len, east), \text{BUILD-RAILING}(x, y, z, len, east)\rangle, \\
[T]\ \text{INS-BUILD-RAILING}(x, y, z, len, east) &\to \langle \mathtt{ins\text{-}teach\text{-}start\text{-}railing}(x, y, z, len, east), \\
&\quad \text{INS-BUILD-BLOCK}(x, y, z), \text{INS-BUILD-BLOCK}(x + len - 1, y, z), \\
&\quad \text{INS-BUILD-ROW}(x + len - 1, y, z, len, west), \\
&\quad \mathtt{ins\text{-}teach\text{-}end\text{-}railing}(x, y, z, len, east)\rangle, \\
&\dots \\
\text{INS-BUILD-BLOCK}(x, y, z) &\to \langle \mathtt{ins\text{-}block(x,y,z)}, \mathtt{put\text{-}block(x,y,z)}\rangle\}
\end{aligned}$$

A	*prec*	*add*	*del*
`put-block(x,y,z)`	{¬ *block(x,y,z)*}	{*block(x,y,z)*}	{}
`ins-block(x,y,z)`	{}	{*lastblock(x,y,z)*}	{*lastblock(x',y',z')* $\mid \forall x', y', z' \in \mathbb{N}$}
`ins-X`	{*knows-T*}	{}	{}
`ins-teach-start-X`	{}	{}	{}
`ins-teach-end-X`	{}	{*knows-T*}	{}

Here, *knows-T* is the feature where T is X's type, e. g. $T = railing$ for $X = \mathrm{railing(x, y, z, length, orientation)}$

Figure 3: Illustration of our construction-planning model (**bold face**) and instruction-planning model.

they allow the modeller to encode different possible instruction variants. Here, we explore this modelling power by allowing each INS-BUILD-X task to decompose in three different ways, corresponding to different levels of abstraction in the explanation of a complex object:

- Decomposition L chooses to explain a complex object in terms of its parts. For instance, it decomposes an "instruct to build railing" task into two "instruct to place block" actions and an "instruct to build row" task. Always choosing Decomposition L results in low-level block-by-block instructions. The plan will alternate `ins-block` actions (instructing the user to "place a block") with `put-block` actions (ensuring correctness through construction planning).
- Decomposition H generates a single instruction action for the object at the current level of abstraction. For the railing, it uses the primitive instruction action `ins-railing` ("build a railing"). This action has a precondition *knows-railing*, i. e. this decomposition can only be chosen if the IF already understands the concept of a railing (either by initial expertise or by previous teaching through decomposition T, see next). The instruction is followed by the construction-planning task BUILD-RAILING ensuring correctness.
- Decomposition T defers instruction to the lower level (like L), but also adds instruction actions `ins-teach-start-X` and `ins-teach-end-X`. These generate utterances like "I will teach you how to build a railing" and have the effect that the IF now understands the corresponding high-level concept (e. g. making the precondition *knows-railing* of `ins-railing` true) so that, later on, decomposition H can be used.

Fig. 4 illustrates how the HTN instruction model interleaves instruction and construction actions to produce valid plans at different levels of abstraction. If Decomposition H is used, the resulting plan will

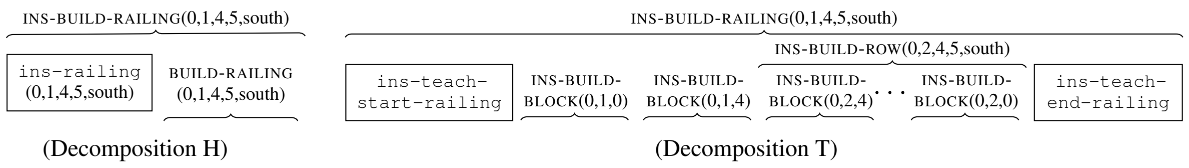


Figure 4: Two example instruction plans for INS-BUILD-RAILING(0,1,4,5,south). Abstract tasks at the bottom layer need to be further decomposed: BUILD-RAILING could be decomposed as in Fig. 2, and INS-BUILD-BLOCK is always decomposed into an `ins-block` and a `put-block` action.

contain a single `ins-railing` instruction action. This will result in sentences such as (c) or (e) in Fig. 1, depending on context. If Decomposition T is used instead, the resulting plan will be much more detailed, with one instruction for every single block that needs to be placed, e. g.(b) in Fig. 1.

5.2 Designing the Cost Function

Given an HTN instruction-planning model as just defined, the HTN planner automatically decides which decomposition methods (in particular: L/H/T) are used at which points of the instruction, in a manner that minimizes plan cost. Instruction quality can thus be optimized by suitably defining the cost function. We demonstrate this by showing how the cost function, in combination with the initial environment state s_0, can be used to realize different instruction-giving *strategies*. In other words, we show how the HTN planner can choose the appropriate level of abstraction. We have realized the following three strategies:

Low-level This strategy always explains how to build complex objects block by block. The initial state does not contain any *knows-X* features, and the cost function assigns a high cost to instruction actions for complex objects (`ins-X` and `ins-teach-X`). Thus, minimal-cost plans always use the L decomposition. (cmp. Fig. 1 a and d)

High-level This strategy always explains how to build complex objects with a single abstract instruction. All *knows-X* state features are true in the initial state, i. e., the IF is assumed to know how to build all complex objects. The `ins-X` actions are assigned low costs, relative to that of `ins-block` actions. Therefore, minimal-cost plans always use the H decomposition. (cmp. Fig. 1 c and e)

Teaching This strategy strikes a balance between the first two. Its initial state assumes that all *knows-X* state features are false; hence, like Low-level, it explains each complex object in simpler terms when it is first built. However, it encourages the use of decomposition T to do this, so that for later instances of the same object decomposition H can be used. This is achieved by assigning a low cost to `ins-X` and `ins-teach-X` relative to the cost of `ins-block`. (cmp. Fig. 1 b and e)

These strategies could, of course, be implemented without the use of a general HTN planning system. However, our generic implementation readily handles deeper user models if available, and it facilitates the flexible combination with other criteria. In particular, the cost of `ins-block` actions can be used to model what the sentence generator can or cannot express easily. In general, this action cost could be determined by the sentence generator, allowing a deep co-optimization between instruction planning and sentence generation (which is future work, see Section 7). For now, we have realized this kind of combination through a simple model reflecting the fact that referring expressions are easier to generate, and to understand, for positions adjacent to the last placed block (which can, for example, be referred to with a pronoun). To this end, we assume a reduced cost for `ins-block` if the placed block is adjacent to the previously placed block (as encoded by the *lastblock* feature).

6 Evaluation

Data collection. We collected evaluation data by asking human subjects to build complex objects in Minecraft under the instruction of our IG system. Study participants were recruited through Prolific. Each participant played a single game, matched to one of the six conditions (three strategies $\times$ two scenarios). Participants were required to be fluent in English and own a Minecraft license. We obtained 20 to 25 plays per condition and paid each participant $\sim$10 GBP per hour.

In each game, the participant was first informed about the target structure ("Welcome! I will try to instruct you to build a [house / bridge]") and then instructed to build this structure until it either was complete or a ten-minute time limit was up and the player had placed at least five correct blocks. We told the participants explicitly that completing the building was not a prerequisite for getting paid, to reduce the risk of people quitting the study because of bad instructions. Either way, the participant was given a secret code word to enter after the game, to guard against cheating.

Each participant filled out a post-experiment questionnaire (see Appendix) after finishing the game. We only considered games for which we also obtained a questionnaire for the evaluation.

Scenarios. We designed two different scenarios to test the different abstraction strategies. The *house* consists of four walls, which are each four blocks wide and two block high, and four rows of four blocks each as the roof. The house is very minimal and has neither a door nor windows. We hypothesized that the high-level strategy would be effective in this scenario, because walls and rows are commonplace objects which the IF may know without needing to be taught.

In the *bridge* scenario, subjects were asked to construct the bridge in Fig. 1, which consists of three complex objects: the floor and two railings. The railings were specifically designed to be of a non-obvious shape for participants. Because of this, we hypothesized that the teaching strategy would work best.

Implementation details. We used the MC-Saar-Instruct platform (Köhn et al., 2020) to connect the IG system to the study participants. They played Minecraft on their own computers and connected to our Minecraft server, which then forwarded their actions to the IG system and the instructions back to the participants. Instruction plans were computed offline, to make the different games comparable and to ensure responsiveness of the IG system (computing a plan usually takes < 1 second but sometimes takes up to 7 seconds). The natural-language instructions were computed online, and changed as a function of the state of the world (e. g., referring expressions used different spatial relations depending on the IF's position in the world). Whenever the IF places a block incorrectly, i. e. in a position that is not consistent with the instruction plan, the IG system needs to guide the IF back on track.[3] In principle, re-planning methods (Ghallab et al., 2004) could be used for this purpose. Here we opted for a simpler solution: the IG system asks the IF to remove that block and then returns to the original plan. A similar heuristic applies when the IF removes a block which was already placed correctly.

Results. In addition to the questionnaire, we also evaluated the IG systems with respect to objective criteria (percentage of successfully constructed buildings, completion time, and number of incorrectly placed blocks). The mean results for each condition are shown in Table 1, including significance test results (Mann–Whitney U test). Below we also report 95% bootstrapped confidence intervals (CI).

6.1 Discussion

Bridge. The IFs took significantly longer to build the bridge under the high-level strategy (mean 275s (CI 217.0, 343.3)) than under the low-level strategy (177s (CI 154.8, 225.6)), and made more mistakes (36.9 (CI 24.3, 55.2) vs. 18.5 (CI 12.3, 28.1)). This is because IFs did not know how to build the very specific railing and thus needed to experiment for a long time. The teaching strategy, which uses high-level descriptions of complex objects only after first explaining them, is as fast and accurate as the low-level strategy (173s (CI 142.7, 226.5)).

At the same time, IFs subjectively rated the low-level strategy significantly lower on the "overall" (2.3 (CI 1.8, 2.7)) and "clarity" (2.4 (CI 1.8, 3.0)) questions than the teaching strategy (overall: 3.4 (CI 2.8, 3.8); clarity: 3.2 (CI 2.7, 3.6)). This confirms our starting hypothesis that low-level instructions can be perceived as tedious by users. Thus the teaching system strikes a good balance of task efficiency and user satisfaction in the bridge scenario.

Looking at the mean building times for the complex subobjects of the bridge, we found that the high-level strategy is much worse for the first railing (low-level 49s; teaching 44s; high-level 129s), but high-level and teaching actually outperform low-level on the second railing (low-level 38s; teaching

[3]While the planner generates a specific reference sequence of put-block actions for each object, the IG accepts every possible order of put-block actions by the user.

	Bridge			House		
	Low-level	Teaching	High-level	Low-level	Teaching	High-level
Success rate (%)	95	100	90	100	95	89
Time to success (seconds)	177.0	172.5○	275.5●●	171.6	239.8◍	244◍○
Number of mistakes made	18.5	18.6○	36.9◍●	14.5	23.3○	29.5◍○
I had to re-read instructions.	4.7	4.1◍	4.6○◍	3.9	4.4○	4.6◍○
It was always clear what to do.	2.4	3.2●	1.9○●	2.8	2.4○	1.9●○
Overall gave good instructions.	2.3	3.4●	2.4○●	2.9	2.6○	2●◍
Gave useful feedback about progress.	3.9	3.8○	4.1○○	4.1	3.9○	4○○
System was really verbose.	2.2	2.8◍	2.2○○	2.2	2.9●	2.9○○
Instructions were too early.	3.8	3.5○	3.1◍○	3.2	3○	3.8○◍
Instructions were too late.	1.4	1.7○	2.1●○	1.4	2●	1.6○◍

Table 1: Means of evaluation results; all data points except "success rate" are only from successful games. Questions results are on a 1-5 scale (1; disagree completely, 5: agree completely). Statements slightly shortened, full text in appendix. Significance levels (Mann-Whitney U) comparing to low-level (for teaching) / low-level and teaching (for high-level) shown as ●: $p < 0.05$, ◍: $p < 0.1$, ○: $p > 0.1$.

18s; high-level 13s). Thus the use of high-level explanations can lead to improved task efficiency, if the complex objects have been explained sufficiently.

House. In the "house" scenario, the high-level strategy still has a significantly higher mean task completion time than the low-level strategy (244s (CI 195.5, 304.2) vs. 171s (CI 152.6, 203.1)) and a higher number of mistakes (29.5 (CI 19.6, 51.3) vs. 14.5 (CI 10.7, 18.4)). This seems to contradict our original hypothesis that IFs should be able to process high-level instructions even without explanation because walls and rows are familiar objects. Furthermore, unlike in the "bridge" scenario, the teaching strategy is slower than the low-level strategy (239s (CI 195.4, 308.0)) and neither judged better overall (low: 2.9 (CI 2.4, 3.5), teach: 2.6 (CI 2.1, 3.1)) nor on clarity (low: 2.8 (CI 2.4, 3.2), teach: 2.4 (CI 2.0, 2.7)). This is puzzling – why should teaching the walls slow the IF down in a way that teaching the railings does not?

To answer this question, we analyzed the building times of the complex subobjects of the house. We find that the mean completion time for the four walls is actually *lowest* for the high-level strategy (76s, compared to low-level 97s, teaching 102s), confirming after all our hypothesis that high-level instructions are efficient for familiar complex objects. Where the teaching and high-level strategy fall behind is the completion time for the first two of the four rows that make up the roof of the house (low-level 53s, teaching 110s, high-level 155s). A closer inspection of the data suggests that this is because the instruction the sentence generator computed for the `ins-row` action, while semantically and syntactically correct, is hard to understand ("build a row to the right of length four to the top of the back right corner of the previous wall"). The low-level block-by-block strategy does not have that problem. Thus, the sentence generator for high abstraction levels must be tested and designed with special care.

7 Conclusion

We have shown how to generate building instructions in Minecraft at different levels of abstraction through HTN planning. Our evaluation shows that the level of abstraction matters to human users, and that our "teaching" strategy strikes an effective balance of high-level and low-level instructions.

We view our system as the first step towards a more general framework. We used a very basic model of the user's knowledge; in future work, we will train statistical models on user interactions to estimate what a user knows initially and what they learn during the experiment. Furthermore, we used hand-designed action costs. Perhaps the most interesting avenue of future research is to let the sentence generator specify the action costs, based on how easy it is to express a given instruction action in language in the current world and dialogue state. This will allow the sentence generator to influence the search of the discourse planner, offering a new method for closing the "generation gap" (Meteer, 1990).

Acknowledgements Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – Project-ID 232722074 – SFB 1102.

References

Ron Alford, Ugur Kuter, and Dana S. Nau. 2009. Translating HTNs to PDDL: A small amount of domain knowledge can go a long way. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1629–1634.

Ron Alford, Pascal Bercher, and David W. Aha. 2015. Tight bounds for HTN planning. In *Proceedings of the 25th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 7–15. AAAI Press.

Krishna Aluru, Stefanie Tellex, John Oberlin, and James MacGlashan. 2015. Minecraft as an experimental world for AI in robotics. In *Proceedings of the AAAI Fall Symposium on Artificial Intelligence and Human-Robot Interaction*, pages 5–12. AAAI Press.

Douglas Appelt. 1985. *Planning English Sentences*. Cambridge University Press.

Gregor Behnke, Daniel Höller, and Susanne Biundo. 2018. totSAT – totally-ordered hierarchical planning through SAT. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence (AAAI)*, pages 6110–6118. AAAI Press.

Gregor Behnke, Daniel Höller, and Susanne Biundo. 2019a. Bringing order to chaos – A compact representation of partial order in SAT-based HTN planning. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAAI)*, pages 7520–7529. AAAI Press.

Gregor Behnke, Daniel Höller, and Susanne Biundo. 2019b. Finding optimal solutions in HTN planning – A SAT-based approach. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 5500–5508. ijcai.org.

Gregor Behnke, Pascal Bercher, Matthias Kraus, Marvin R. G. Schiller, Kristof Mickeleit, Timo Häge, Michael Dorna, Michael Dambier, Dietrich Manstetten, Wolfgang Minker, Birte Glimm, and Susanne Biundo. 2020. New developments for robert – Assisting novice users even better in DIY projects. In *Proceedings of the 30th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 343–347. AAAI Press.

Pascal Bercher, Gregor Behnke, Daniel Höller, and Susanne Biundo. 2017. An admissible HTN planning heuristic. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 480–488. ijcai.org.

Pascal Bercher, Ron Alford, and Daniel Höller. 2019. A survey on hierarchical planning – One abstract idea, many concrete realizations. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 6267–6275. ijcai.org.

Kutluhan Erol, James A. Hendler, and Dana S. Nau. 1994. HTN planning: Complexity and expressivity. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI)*, pages 1123–1128. AAAI Press.

Konstantina Garoufi and Alexander Koller. 2014. Generation of effective referring expressions in situated context. *Language, Cognition, and Neuroscience*, 29(8):986–1001.

Malik Ghallab, Dana S. Nau, and Paolo Traverso. 2004. *Automated planning – Theory and Practice*. Elsevier.

Jonathan Gray, Kavya Srinet, Yacine Jernite, Haonan Yu, Zhuoyuan Chen, Demi Guo, Siddharth Goyal, C Lawrence Zitnick, and Arthur Szlam. 2019. Craftassist: A framework for dialogue-enabled interactive agents. arXiv preprint arXiv:1907.08584.

Daniel Höller, Pascal Bercher, Gregor Behnke, and Susanne Biundo. 2020. HTN planning as heuristic progression search. *Journal of Artificial Intelligence Research*, 67:835–880.

Eduard Hovy. 1988. *Generating natural language under pragmatic constraints*. Lawrence Erlbaum Associates, Hillsdale, NJ.

Srinivasan Janarthanam and Oliver Lemon. 2010. Learning to adapt to unknown users: Referring expression generation in spoken dialogue systems. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 69–78. The Association for Computer Linguistics.

Arne Köhn and Alexander Koller. 2019. Talking about what is not there: Generating indefinite referring expressions in Minecraft. In *Proceedings of the 12th International Conference on Natural Language Generation (INLG)*, pages 1–10. Association for Computational Linguistics.

Arne Köhn, Julia Wichlacz, Christine Schäfer, Álvaro Torralba, Joerg Hoffmann, and Alexander Koller. 2020. MC-Saar-Instruct: a platform for minecraft instruction giving agents. In *Proceedings of the 21th Annual Meeting of the Special Interest Group on Discourse and Dialogue*, pages 53–56, 1st virtual meeting, July. Association for Computational Linguistics.

Alexander Koller, Kristina Striegnitz, Donna Byron, Justine Cassell, Robert Dale, Johanna Moore, and Jon Oberlander. 2010. The first challenge on generating instructions in virtual environments. In E. Krahmer and M. Theune, editors, *Empirical Methods in Natural Language Generation*, volume 5790 of *LNAI*. Springer.

Marie Wenzel Meteer. 1990. *The "GENERATION GAP": The Problem of Expressibility in Text Planning*. Ph.D. thesis, USA. UMI Order No. GAX90-22720.

Anjali Narayan-Chen, Prashant Jayannavar, and Julia Hockenmaier. 2019. Collaborative dialogue in Minecraft. In *Proceedings of the 57th Conference of the Association for Computational Linguistics (ACL)*, pages 5405–5415. Association for Computational Linguistics.

Dana S. Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, J. William Murdock, Dan Wu, and Fusun Yaman. 2003. SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research*, 20:379–404.

Priyam Parashar, Bradley Sheneman, and Ashok Goel. 2017. Adaptive agents in Minecraft: A hybrid paradigm for combining domain knowledge with reinforcement learning. In *Proceedings of the AAMAS Workshop on Adaptive Learning Agents (ALA)*.

Mark Roberts, Wiktor Piotrowski, Pyrce Bevan, David Aha, Maria Fox, Derek Long, and Daniele Magazzeni. 2017. Automated planning with goal reasoning in Minecraft. In *Proceedings of the ICAPS Workshop on Integrated Execution of Planning and Acting (IntEx)*, pages 43–49.

Roan Boer Rookhuiszen, Michel Obbink, and Mariët Theune. 2009. Two approaches to give: dynamic level adaptation versus playfulness. In *Proceedings of the First Challenge on Generating Instructions in Virtual Environments (GIVE)*.

Julia Wichlacz, Daniel Höller, Álvaro Torralba, and Jörg Hoffmann. 2020. Applying Monte-Carlo Tree Search in HTN planning. In *Proceedings of the 13th International Symposium on Combinatorial Search (SoCS)*, pages 82–90. AAAI Press.

A The Post Game Questionnaire

- Overall, the system gave me good instructions.
- I had to re-read instructions to understand what I needed to do.
- It was always clear to me what I was supposed to do.
- The system's instructions came too late or too early.
- The system was really verbose and explained things that were already clear to me.
- The system gave me useful feedback about my progress.
- Please add any comments or observations you had (free text)

All but the last question are five-point Likert scale questions (Disagree completely – Agree completely).

B Build Times for the Bridge Scenario

	Low-level	Teaching	High-level
floor	84.1 (CI 71.6 100.5)	103.9 (CI 78.5 164.5)	128.9 (CI 91.6 190.5)
railing 1	49.4 (CI 40.3 68.2)	44.3 (CI 37 53.1)	129.9 (CI 92.4 190.4)
railing 2	38.2 (CI 24.3 79.1)	18.4 (CI 10.7 47.4)	13.3 (CI 9.6 21.6)

C Build Times for the House Scenario

	Low-level	Teaching	High-level
wall 1	51.9 (CI 46.3 61.2)	68.6 (CI 59.1 78.2)	52.9 (CI 39.9 68.1)
wall 2	23.2 (CI 18 34.4)	15.2 (CI 11.7 24)	10.3 (CI 8.1 16.3)
wall 3	12.7 (CI 10.5 16.1)	8.6 (CI 6.5 11.9)	6.7 (CI 5.4 10.1)
wall 4	9.4 (CI 8.1 10.8)	9.5 (CI 6 17.4)	5.8 (CI 4.5 7.6)
row 1	17.1 (CI 14 22.7)	42.7 (CI 32.3 68.4)	69.6 (CI 52.5 95.1)
row 2	35.5 (CI 21.5 61.5)	66.9 (CI 38.7 124.6)	85.2 (CI 48.7 153.9)
row 3	9.0 (CI 6.2 14.5)	8.6 (CI 5.8 17.6)	7.9 (CI 4.1 18.1)
row 4	5.7 (CI 4.3 9.9)	13.9 (CI 4.9 44.6)	3.8 (CI 2.8 5.2)

D Planning Times for the different Configurations

Scenario	Low-level	Teaching	High-level
Bridge	0.78s	0.81s	0.82s
House	5.92s	0.96s	6.98s