# 4 Proposed Research

To illustrate concretely what we mean by learning, doing and teaching based on a unified theory, we have implemented a simple proof of concept system, which deals with only a few structural features of hierarchical tasks. Following a sequence of interaction walkthroughs with this system presented below, we discuss the overall architecture of our unified theory, and then detail our specific proposed research goals and timeline, including the Learn-Do-Teach Challenge.

## 4.1 The Learn-Do-Teach Proof of Concept System

*Hierarchical task networks* (HTNs) are a common and convenient representation for procedural knowledge about hierarchical tasks. Our proof of concept system was implemented in Disco (see Section 2.1), which uses the ANSI/CEA-2018 standard [43] for HTNs, whose definition was led by PI Rich. In order to explain the interaction walkthroughs below, we first need to introduce some technical terminology and notational conventions.

The diagram on the right side of Figure 3 shows the simple example HTN used in all the walkthroughs. To avoid distracting details, the tasks in this model have abstract names, such as A, B and C. Figure 10 shows a more realistic task model in the car maintenance domain.

An HTN is essentially a set of task decomposition rules, commonly called *recipes*, each of which (e.g., r1 in Figure 3) decomposes a *non-primitive* task (e.g., A) into a sequence of primitive and/or non-primitive tasks (e.g., B and C), which are called the *steps* of the recipe. In this proof of concept system, all recipes are totally ordered, as indicated by the arrows between the steps. Additional features of HTNs in ANSI/CEA-2018, such as partially ordered recipe steps, preconditions, postconditions, recipe applicability conditions and constraints will be discussed in Section 4.3.1.

The complete set of recipes that recursively decompose a given non-primitive task, such as A, into *primitive* tasks, such as d, e, f, g, h and i, is conventionally drawn as a tree, such as in Figure 3, where the primitives are at the fringe. The dotted lines in Figure 3 indicate that r2 and r3 are two alternative recipes for (ways to decompose) B. This structure is sometimes called an "and-or tree."

As a typographical convention in this document, the names of non-primitive tasks will always start with an uppercase letter, e.g., A or LearnStep, and the names of primitive tasks will start with a lowercase letter, e.g., d or addStep. The rest of this section contains seven interaction walkthroughs illustrating how our proof of concept system passes the Learn-Do-Teach Challenge for the simple HTN shown in Figure 3. The Learn-Do-Teach Challenge is described in more detail in Section 4.4.

### 4.1.1 Doing Hierarchical Tasks Collaboratively

The first walkthrough, shown on the left side of Figure 3, serves as an introduction to SharedPlan theory, the Disco collaboration manager and the notation, called a *segmented interaction history*, that Disco uses to record the structure of a collaborative interaction. With a few small changes (such as bold fonts) to improve readability, all of the text shown in Figure 3, including the utterance glosses in quotes, is automatically generated by Disco as the interaction proceeds.

In this walkthrough, all the recipes in the HTN in Figure 3 (r1, r2, r3 and r4) are already known to both participants in the collaboration, i.e., a human and an agent implemented in Disco. This is thus an example of the *do* component of learn-do-teach. This example also, however, lays the foundation for how to computationally model teaching/learning as a kind of collaboration.

Since collaboration requires communication in some form (verbal and/or nonverbal), collaborative interactions are also dialogues. In this document we will therefore use the terms *dialogue* and *interaction* interchangeably to refer to interactions that, like Figure 3, include a mixture of actions and utterances.

Notice first that an interaction history is hierarchical (shown by indentation) and that each level of the hierarchy, called a *segment*, is introduced by a line in square brackets (e.g., lines 1, 3, 7, and 9) that indicates the *purpose* (goal) of the following segment. Thus the purpose of the toplevel segment is to achieve task A, with two subsegments to achieve B and C, and so on. In addition, for segments whose purpose is to achieve a non-primitive task, the recipe used is also named (e.g., "by r1"). The fundamental insight of SharedPlan dialogue theory, which we see in this example, is that in task-oriented dialogues, the interaction structure reflects the underlying task structure.[1] Each line in the segmented interaction history other than the bracketed

[1]This does not mean that the interaction structure is always identical to the task structure. It is possible for the focus of attention to shift around in the task (see [30]).

```
1  [A by r1]
2     do(human,A)                    "Let's do A."
3     [B by r2]
4        d(agent)
5        e(agent)
6        done(agent,B)               "We're done doing B."
7     [C by r4]
8        h(human)
9        [i(agent)]
10          do(human,i(agent))       "Please do i."
11          i(agent)
12       done(human,C)               "We're done doing C."
13    done(human,A)                  "We're done doing A."
```

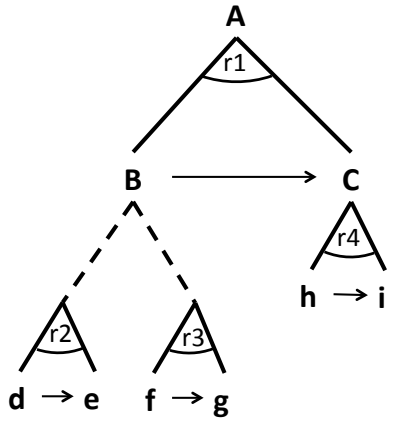


Figure 3: Segmented interaction history for a collaborative execution of example HTN.

purpose lines corresponds to an event in the interaction, which is the occurrence of either a primitive task (action) from the domain model, such as **d(agent)** on line 4, or an utterance, such as do(**human**,A) on line 2. Utterances are also primitive tasks. Primitive tasks are shown in the history as operators, where the first parameter indicates who performed the task (e.g., the agent or human) and additional parameters depend on the type of task. This operator notation could be further formalized in a logic of action [35], but that is not focus of this work.

The only two utterances used in this proof of concept system ('do' and 'done') are defined in the generic layer of our theory (see Section 4.1.5), and do not appear in application-specific recipes, such as the HTN in Figure 3. These utterances are dynamically generated by the Disco agent according to SharedPlan theory [44]. Furthermore, both of these utterances are *meta* (higher-order) operators, because they take an operator as a parameter. The utterance do(**human**,A), said by the human on line 2, is glossed as "Let's do A" and signals the start of a new segment whose purpose is to achieve A. The utterance done(**human**,A) on line 13, which is glossed as "We're done doing A," signals the end of that segment. When 'do' is used with primitive tasks, the performer of the target task must also be specified,[2] as in do(**human**,i(agent)) on line 10, where the human tells the agent to "Please do i," which the agent subsequently does on line 11. The 'done' utterance is optional and is often omitted for segments whose purpose is a primitive task.

The agent's utterances and actions in this interaction were automatically chosen and performed by Disco; the human's utterances and actions were selected from a menu by a human. A slightly different interaction would have resulted if the human made different choices. For example, in line 10 the human could have chosen to perform i herself rather than asking the agent to do it.

### 4.1.2 Pedagogical Strategies as Meta-Recipes

Now, how can we model the goal of a *learning* interaction in which, for example, a human teaches an agent how to do the non-primitive task C? In general, the desired result of such an interaction is for the learner to have a mental representation at least sufficient to perform the task being taught. We therefore model the goal of such a teaching interaction as a new generic non-primitive task LearnRecipe(C,r4,agent). LearnRecipe is a *meta-task*, because it takes a task as a parameter. The postcondition (success condition) of LearnRecipe(C,r4,agent) is for the agent to have a mental representation equivalent to recipe r4 for C.

To carry out this approach, we also need to introduce generic primitives that operate on the learner's mental representation of recipes. In our proof of concept system, we have only one such primitive, namely addStep(?learner,?task,?recipe), which changes the mental state of the learner by adding the given task as the next step in the given recipe. Notice that addStep is a primitive meta-task because it takes a task as a parameter.

So, what about recipes for LearnRecipe? We call these *pedagogical strategies*. For example, in our proof of concept system, we have only one strategy for teaching the steps of a recipe, which is to teach them in order. This recipe, called *steps*, is shown at the top of Figure 4 and has a variable number of steps corresponding to the number of steps in the recipe being taught. We call this a *meta-recipe* because it decomposes a meta-task.

Notice that Figure 4 introduces an additional feature of HTNs, namely, *constraints* (bindings) between the parameters of tasks. In the notation used here, this is indicated by the use of the same variable name, e.g., *?recipe*, in several places in the same recipe.

The steps of the recipe for LearnRecipe are of type LearnStep. LearnStep has three recipes: instruction, demonstration and topdown. Notice that the execution of LearnStep always ends with performing addStep. We will see examples of using each of these recipes and explain them further in the walkthroughs below.

[2] In SharedPlan theory, all shared non-primitive goals are the joint responsibility of both collaborators, regardless of whether all of the primitives in a recipe happen to be performed by only one collaborator.

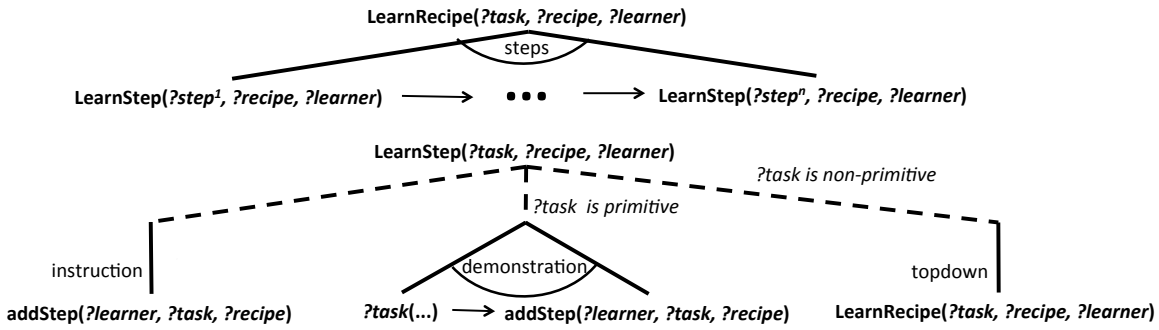


Figure 4: Pedagogical meta-recipes.

In order to restrict when each of the recipes for LearnStep may be used, Figure 4 introduces yet another feature of HTNs, which is the option to associate a boolean *applicability condition* with a recipe. For example, the instruction recipe has no applicability condition, which means it may always be used. However, the demonstration recipe may only be used when the task of LearnStep is primitive; the topdown recipe may only be used when the task of LearnStep is non-primitive.

More than one recipe for a task may be applicable in a given situation. For example, both instruction and demonstration may be used for any given primitive step. This is a choice that the teacher is free to make and may be informed by application-specific and/or generic heuristics. For example, a generic heuristic might be to always use the demonstration recipe the first time a given primitive is used, and then to use the instruction recipe thereafter.

To summarize, we have used *reflection* to unify collaborative learning, doing and teaching. Because the meta-recipes are represented the same way as application-specific recipes, all of the computational mechanisms of collaboration can be uniformly applied at both the object (application-specific) and meta (generic pedagogical) levels, making it possible to seamlessly shift between learning, doing and teaching. In Section 4.3.3, we discuss applying reflection one more time to *learn* pedagogical strategies (meta-recipes) within the same unified theory.

### 4.1.3 Learning from Instructions

Figure 5(a) illustrates the use of the *instruction* recipe for LearnStep in Figure 4 by the human to teach the agent how to do C. The instruction recipe for LearnStep captures (in a simple way in this proof of concept system) the pedagogical strategy of "telling" someone how to do something.

The interaction begins on line 2, where the human introduces the LearnRecipe goal in exactly the same way that the A goal was introduced on line 2 in Figure 3. Next, on line 4, the human implicitly chooses[3] the instruction recipe for the first LearnStep of LearnRecipe by telling the agent to perform the appropriate addStep. This utterance is glossed as "First step is h," since r4 doesn't have any steps yet.

In line 5, the agent performs the second step of the instruction recipe, addStep, which appropriately updates its mental representation of r4. Lines 7 and 8 follow similarly for the second step of r4. Finally, on line 9, the human communicates the fact that are no more steps in r4.

Next, in Figure 5(b), after being asked by the human, the agent shows that it has learned how to do C by correctly performing all the steps in recipe r4.

### 4.1.4 Learning from Demonstration

Figure 5(c) illustrates the use of the *demonstration* recipe for LearnStep in Figure 4 by the agent above to teach *another copy of itself* (`agent'`) how to do C. (The other copy obviously does not already know how to

[3]Disco recognizes this choice via plan recognition [29].

```
[LearnRecipe(C,r4,agent) by steps]
   do(human,LearnRecipe(C,r4,agent))                "Let's learn how to do C."
   [LearnStep(h,r4,agent) by instruction]
      do(human,addStep(agent,h,r4))                 "First step is h."
      addStep(agent,h,r4)
   [LearnStep(i,r4,agent) by instruction]
      do(human,addStep(agent,i,r4))                 "Next step is i."
      addStep(agent,i,r4)
   done(human,LearnRecipe(C,r4,agent))              "We're done learning how to do C."
```

(a) Human teaching agent by instruction how to do C.

```
[C by r4]
   do(human,C)                                      "Let's do C."
   h(agent)
   i(agent)
   done(agent,C)                                    "We're done doing C."
```

(b) Agent doing C.

```
[LearnRecipe(C,r4,agent') by steps]
   do(agent,LearnRecipe(C,r4,agent'))               "Let's learn how to do C."
   [LearnStep(h,r4,agent') by demonstration]
      h(agent)
      addStep(agent',h,r4)
   [LearnStep(i,r4,agent') by demonstration]
      i(agent)
      addStep(agent',i,r4)
   done(agent,LearnRecipe(C,r4,agent'))             "We're done learning how to do C."
```

(c) Agent teaching another copy of itself by demonstration how to do C.

```
[LearnRecipe(C,r4,human') by steps]
   do(agent,LearnRecipe(C,r4,human'))               "Let's learn how to do C."
   [LearnStep(h,r4,human') by demonstration]
      h(agent)
      addStep(human',h,r4)
   [LearnStep(i,r4,human') by instruction]
      do(agent,addStep(human',i,r4))                "Next step is i."
      addStep(human',i,r4)
   done(agent,LearnRecipe(C,r4,human'))             "We're done learning how to do C."
```

(d) Agent teaching another human by mixture of demonstration and instructions how to do C.

Figure 5: Implemented proof of concept system illustrating Learn-Do-Teach Challenge and mixture of learning from demonstration and instructions.

do C.) The demonstration recipe for LearnStep captures a simple version of learning from demonstration, wherein after a primitive task is performed, the learner adds the task to its mental representation of the current recipe.

This interaction begins on line 16 as usual with the introduction of the toplevel goal. Then on line 18, the teaching agent implicitly chooses the demonstration recipe for the first LearnStep of LearnRecipe by performing h. On line 19, the learning agent finishes execution of the demonstration recipe by performing addStep with the appropriate parameter bindings, thereby updating its mental representation of r4. Lines 21 and 22 follow similarly for the second step of r4. Finally, on line 23, the teaching agent communicates the fact that are no more steps in r4.

Figure 5(d) illustrates how the hierarchical structure of the pedagogical meta-recipes allows the intermixing of demonstration and instruction in teaching a single application-specific recipe. In this interaction, the original agent teaches *another human* (**human'**) how to do C, choosing demonstration for the first step and instruction for the second step of r4. Notice on lines 28 and 31 that, since the learner here is a human, the proof of concept system simply assumes she has performed addStep to update her mental representation of r4 (see comment at the end of Section 4.1.5 regarding grounding).

In the four interactions above, our proof of concept agent has passed a simple version of the Learn-Do-Teach Challenge! It has (a) learned how to do C from a human, (b) done C, (c) taught another copy of itself how to do C, and (d) taught another human how to do C.

### 4.1.5 More Learning

To finish learning the complete HTN in Figure 3, we need two more interactions. Figure 6 shows how, after the agent already knows how to do C, the human teaches it how to do A. This interaction illustrates the use of the *topdown* recipe for LearnStep in Figure 4, which has a single step that is a mutually-recursive invocation of LearnRecipe. The topdown recipe for LearnStep captures the pedagogical strategy of starting with the topmost task in a hierarchy and incrementally decomposing it—versus starting with primitives and combining them into bigger and bigger non-primitives (as was done in Figure 5(a)). These two strategies can obviously be intermixed.

The interaction starts as usual on line 2 with the introduction of the toplevel goal, which is to learn the recipe for A. Then on line 5, the human implicitly chooses the topdown recipe for the first LearnStep of LearnRecipe by introducing `LearnRecipe(B,r2,agent)` as a subgoal. Then, in the remainder of this

```
1  [LearnRecipe(A,r1,agent) by steps]
2     do(human,LearnRecipe(A,r1,agent))                  "Let's learn how to do A."
3     [LearnStep(B,r1,agent) by topdown]
4        [LearnRecipe(B,r2,agent) by steps]
5           do(human,LearnRecipe(B,r2,agent))           "Let's learn how to do B."
6           [LearnStep(d,r2,agent) by demonstration]
7              d(human)
8              addStep(agent,d,r2)
9           [LearnStep(e,r2,agent) by demonstration]
10             e(human)
11             addStep(agent,e,r2)
12          done(human,LearnRecipe(B,r2,agent))  "We're done learning how to do B."
13    [LearnStep(C,r1,agent) by instruction]
14       do(human,addStep(agent,C,r1))                          "Next step is C."
15       addStep(agent,C,r1)
16    done(human,LearnRecipe(A,r1,agent))        "We're done learning how to do A."
```

Figure 6: Human teaching agent how to do A.

```
1  [LearnRecipe(B,r3,agent) by steps]
2     do(human,LearnRecipe(B,r3,agent))          "Let's learn another way to do B."
3     [LearnStep(f,r3,agent) by demonstration]
4        f(human)
5        addStep(agent,f,r3)
6     [LearnStep(g,r3,agent) by demonstration]
7        [g(agent)]
8           do(human,g(agent))                              "Please do g."
9           g(agent)
10       addStep(agent,g,r3)
11    done(human,LearnRecipe(B,r3,agent))        "We're done learning how to do B."
```

Figure 7: Human teaching agent another way to do B.

segment, the human proceeeds to teach the steps of B by demonstration, similar to Figure 5(b). In the next subsegment, starting at line 13, the human simply tells the agent that the next step of the recipe for A is C, since the agent already knows how to do C.

Now the only knowledge in Figure 3 remaining untaught is the second (alternative) recipe for B, namely r3. The human teaches this recipe to the agent in Figure 7. The interaction starts as usual on line 2 with the introduction of the toplevel goal, which is glossed as "Let's learn another way to do B," because there is already one known recipe for B. The first step of r3 is taught by demonstration, similarly to previous uses of the demonstration recipe. However, there is a small variation here in the way that the second step of r3 is taught by demonstration, which illustrates the generality of the pedagogical recipes. In line 8, instead of performing g herself, the human asks the agent to perform g, which it subsequently does on line 9. The agent then finishes execution of the demonstration recipe by performing addStep with the appropriate bindings.

This completes the interaction walkthroughs with our proof of concept system. All of the segmented interaction histories shown were automatically generated by our implementation. In order to focus on the learning semantics, we have somewhat simplified these interactions by omitting grounding behaviors (such as saying "Ok") and negotiation (such as accepting or rejecting a proposed goal), which are supported by Disco and which we plan to use in our proposed research. In the following section, we step back and look at the conceptual architecture of our unified theory approach.

## 4.2 Architecture of Unified Theory

Figure 8 shows the fours layers in the conceptual architecture of our unified theory of learning, doing and teaching hierarchical tasks. The bottom two layers are application-specific, while the top two layers are generic. Each layer shows examples from our proof of concept system.

The bottommost layer in the architecture is the *domain model*, which reflects fundamental decisions about how to represent the possible states of the application world. Because domain modeling is not the focus of this work, we adopt here a simple and commonly used approach, in which the domain model consists of a set of predicates that describe the state of the world and operators that change that state. In our proof of concept system, the operators are the primitive tasks: d, e, f, g, h

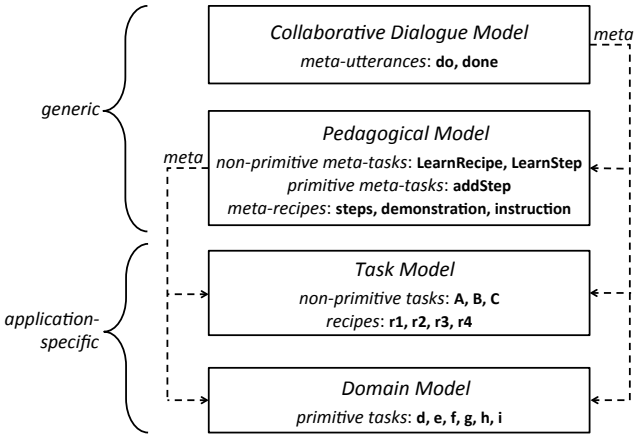


Figure 8: Layers of unified theory with examples from proof of concept system.

and i. We didn't define any predicates in our proof of concept system, but in general, world state predicates are needed to specify preconditions and postconditions (see Section 4.3.1) of tasks. It is also possible for there to be application-specific utterances. Other researchers have worked on learning domain models [18, 70, 72].

The second layer in the architecture, called the *task model*, contains the procedural knowledge that enables a human or agent to expertly perform and communicate about complex application tasks. Teaching and learning tasks models through collaborative interaction is the focus of this research. As discussed above, we have chosen HTNs as a common and convenient representation of this knowledge. Thus our task model contains the definitions of non-primitive application tasks and the recipes for them, as for example shown in Figure 3.

The *pedagogical model* is the heart of our approach. The simple pedagogical model in our proof of concept system is shown in Figure 4. Refining and expanding this pedagogical model is one of the major goals of our research. As discussed above, we use reflection to express generic pedagogical strategies as meta-recipes in the same HTN representation as the task model. Using the same representation and processes at both the object and meta levels is what makes our theory unified with respect to learning, doing and teaching.

Finally, the *collaborative dialogue model* contains generic meta-utterances, such as 'do' and 'done' in our proof of concept system, that are used to mediate the collaboration process. Notice that these meta-utterances can be applied to either pedagogical or application-specific tasks. Disco contains a number of additional such meta-utterances, for example to repeat or stop working on a specified task [49], which we expect to take advantage of in our further research.