

Scaffolding Instructions to Learn Procedures from Users

Paul Groth and Yolanda Gil

Information Sciences Institute
University of Southern California
pgroth@isi.edu , gil@isi.edu

Abstract

Humans often teach procedures through tutorial instruction to other humans. For computers, learning from natural human instruction remains a challenge as it is plagued with incompleteness and ambiguity. Instructions are often given out of order and are not always consistent. Moreover, humans assume that the learner has a wealth of knowledge and skills, which computers do not always have. Our goal is to develop an electronic student that can be made increasingly capable through research to learn from human tutorial instruction. Initially, we will provide our student with human-understandable instruction that is extended with many scaffolding statements that supplement the limited initial capabilities of the student. Over time, improvements to the system are driven and quantified by the removal of scaffolding instructions that are not considered to be natural for users to provide humans. This paper describes our initial design and implementation of this system, how it learns from scaffolded instruction in two different domains, and the initial relaxations of scaffolding that the system supports.

Introduction

Humans learn procedures from one another through a variety of methods, such as observing someone do the task, watching many examples, and reading manuals or textbooks. Examples are a very natural way to teach, but when procedures are complex it becomes unmanageable to induce the procedure from examples. A very common method is tutorial instruction, which describes in general terms what actions to perform and possibly includes explanations of the dependencies among the actions. These different methods are often combined, where instruction is followed by illustrative examples, or examples are complemented with instruction that guides the induction process. Researchers have investigated methods to learn procedures from examples (Kaelbling, Littman, and Moore 1996; Yang, Wu, and Jiang 2007; Lieberman 1994; Lau et al. 2003; Winner and Veloso 2002), sometimes in combination with some form of instruction given as advice (Maclin and Shavlik 1996) or explanation (Allen et al. 2007).

Copyright © 2009, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

A different line of research has investigated learning procedures from user instruction, focusing on learning modifications to existing procedures (Blythe et al. 2001) or assembling procedures from pre-existing tasks (Clark et al. 2001; Kim and Gil 2001). These approaches require the user to provide the instruction in the form required by the system. Other approaches to learning from instruction are more natural for users as they are free to provide any kind of instruction and can do so in natural language (Webber et al. 1995; Huffman and Laird 1995).

Instruction can be of many kinds (Webber et al. 1995), including general policies regarding acceptable behaviors, advice on how to proceed, suggestions for preferences, analogies, requests, and tutorial descriptions. Another important factor is whether it is given before, during, or after the procedure takes place. Our initial focus is on tutorial instruction given before execution or planning takes place.

Our broad goal is to learn procedures from human instruction stated in a manner that is natural to provide. Although all humans are natural teachers, and instruction is a very common way to teach, humans are often poor instructors. Anyone who has consulted a manual for programming a VCR can attest to this. Studies have shown that better instruction enables better and faster learning. In spite of poor instruction, humans often times learn by compensating with expertise or practice.

Acknowledging the many challenges of developing electronic students and the many contributions of prior research in this topic, we focus on a novel area of research: a new approach to the design electronic students that can take on increasingly more deficient instruction (from the point of view of ease of learning) as they are extended over time with new capabilities. The strategy to develop these systems is to start designing them to take on complete and correct instruction both in content and organization to facilitate the student's learning. As learning by instruction continues, some of the benevolent properties of the instruction are removed and the student is extended to accommodate those deficiencies. Eventually, the student should accept instruction in a form that is natural for humans to provide. In this paper, we show our initial work in developing this approach for learning procedures from tutorial instruction. The contributions of this paper are as follows:

1. A framework for building electronic students for learning

from tutorial instruction.

2. A symbolic execution algorithm for determining the plausibility of a procedure description.
3. An initial implementation of the framework that generates learned procedures in a variety of formats.

The paper is structured as follows. We begin with an overview on human instruction and the motivation and focus for this research. We then discuss a general framework for learning from tutorial instruction. We describe our target procedure language and the language used to provide instructions, followed by examples from RoboCup and Solitaire of incomplete tutorial instruction. We then present the algorithms, heuristics and their implementation that enables learning procedures from these instruction. Finally, we present the results generated by our implementation and conclude.

Motivation

Human instruction is ubiquitous and yet is often poor and leads to inefficient or little learning. The order of instructions is important. Human learners process much better instructions that provide first an overview of the organization of the procedure and then details on each of the steps [Dixon 87]. When steps are provided first, subjects attempted immediately to interpret them but then had to correct their interpretation once the organizational portion of the instruction was given. When the organizational information is provided first, subjects formed a structure where the component steps could be directly assimilated. Omissions in the instructions are also an important factor. Human learners assume that steps that are important to the procedure will be explicitly stated in the instruction, though they manage to learn when steps are implicit in the instruction (Dixon, Faries, and Gabrys 1988). Conversely, when steps are relatively unimportant but stated explicitly, the initial inferences made about them tend to be erroneous and must be later corrected. This does not happen when learners have expertise in the subject matter, and use their own judgment to decide on the importance of the steps. Another important factor is the overall structure of the instructions. Poor instruction results when its structure does not parallel the hierarchical structure of the procedures being described [Donin et al 92]. Finally, the amount of information or detail specified affects comprehension. The absence of information about action dependencies and about how a device works hinders the learning of new procedures (Steehouder, Karreman, and Ummelen 2000). (Mahling and Croft 1989) found that most people are very good at expressing task decomposition, sequencing, and preconditions, but are not very good at recalling effects of procedures and actions.

Given that humans often provide deficient instruction, our goal is to develop an electronic student that can be resilient to such deficiencies. Our research is driven to supplement particular deficiencies in instruction that we will introduce as the student is capable of handling them. These deficiencies result from both the teacher giving less than ideal (but natural) instruction, and the student capabilities and prior knowledge being less than ideal for learning from less than

ideal instruction. This is related to what is known in tutoring systems as tutorial scaffolding (Reiser 2004), where a teacher adds extra information to facilitate learning when the student is a poor learner or the material is too advanced. Scaffolding is added (deepening) or removed (fading) over time as the student is found to need it or no longer need it respectively. In our case, the extra information included in the instruction may be seen as scaffolding, as our student is not yet capable of absorbing instruction when this information is absent. Eventually, to become a good student, our student will also have to accommodate poor teaching.

Framework

We now present a three-stage framework shown in Figure 1 for learning from tutorial instruction. The aim of this framework is to support the investigation of algorithms to deal with increasingly deficient instruction. It is not designed to deal directly with natural language instruction. Instead, it uses structured markup, which allows for the systematic investigation of instruction ambiguities. The framework is designed to accommodate new learning capabilities within the student. We assume that the student has some prior knowledge that forms its initial knowledge base before instruction.

Ingestion

The first stage of the framework is ingestion, where instruction markup is converted into a set of assertions about the procedure that we call a *procedure stub*. During ingestion, the system identifies subtasks, their ordering, the dependencies between subtasks, the state required to execute each subtask, and subtask parameters. Ingestion is done independently from any prior knowledge of the student. This enables a separation of concerns between the conversion of instruction markup and the student's internal reasoning. In very few cases will tutorial instruction result in a complete, self-contained procedure stub that is sufficient for its execution. Ingestion simply creates a set of assertions about the procedure, and does not attempt to rectify incomplete or imprecise instruction. Instead, these gaps in ingested knowledge are rectified in the next stage of the framework.

Elaboration

Elaboration consists of two parts: the mapping of ingested knowledge with student's prior knowledge and the application of reasoning to infer missing process knowledge. The mapping of knowledge from multiple sources is a complex topic. Our initial framework adopts a simplified approach. It assumes that the teacher and the student use a common vocabulary of predicates and properties across lessons. The result of mapping is a set of *expanded procedure stubs*. Currently, because of our mapping assumption only one expanded procedure stub is produced. After mapping, the elaboration step takes these expanded procedure stubs and applies heuristics that attempt to fill in the gaps in the procedural knowledge by formulating hypotheses about those gaps. The application of these heuristics results in the generation of alternative hypotheses for the procedure or *procedure hypotheses*. The student must then be able to deter-

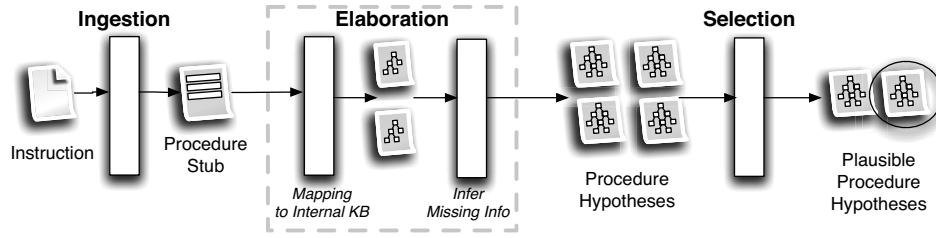


Figure 1: A framework for learning from tutorial instruction

mine which hypotheses are inconsistent and which may be successful during execution, which is done next.

Selection

At this stage, the student must decide which procedure hypotheses to discard given what it knows. To do this, we adopt an approach from program testing and analysis, namely, symbolic execution (King 1976). Instead of executing a program with inputs initialized to particular values, the inputs in symbolic execution are given symbolic values, which represent a class of inputs. Thus, the consistency of the program for that class of inputs is checked. Symbolic execution is similar to the static evaluation techniques proposed in (Etzioni 1993) but it starts with a high-level description of a goal. The exact algorithm used for symbolic execution is described later in the Selection Algorithm section.

Broadly, the student starts from a nominal state formed by nominal objects of the type required to initiate the procedure hypothesis (Skoelms), the student then walks through the decomposition network of the procedure candidate. When a primitive task is found the system returns additional nominal objects as needed. During the walk, if multiple procedures are found the algorithm branches investigating each possibility. If at any point the state is inconsistent within a branch, that branch discontinues execution. Likewise, if a task (either primitive or another procedure) required by the decomposition is not found, the branch discontinues execution. If all the branches of execution for a procedure hypotheses are discontinued, then the hypothesis is ruled out. The result of this process is a set of *plausible procedure hypotheses*.

As the execution is performed, a trace is also created. The trace contains all the procedures and primitive tasks that were executed, the branches of decomposition, and the symbols used during execution. This trace enables plausible procedure hypothesis to be compared and ranked.

Procedure Language

Instructions, as in many technical and scientific expositions, are often provided with a goal-oriented hierarchical structure (Britt and Larson 2003). Representing procedures as goal decomposition hierarchies has also been found to be useful for providing explanations of problem solving behavior (Swartout, Paris, and Moore 1991). The same representations are more amenable to be modified by users (Gil

and Melz 1996). In addition, hierarchical goal decomposition is used in many planners and solvers (e.g., (Nau et al. 2003)). Consistent with this body of research, we adopt a representation of procedures that decomposes them into a set of subtasks to be accomplished by other procedures as well as primitive tasks that can be directly executed in the world.

The language represents procedures as objects with properties that relate the procedures to other entities or objects. Through the learning process, the student makes assertions about the procedures. Each *procedure* has a *signature* which defines its *name*, a set of conditions indicating the state features or *inputs* that the procedure needs in order to be applicable, and, possibly, the *result* of executing the procedure. Both inputs and results can be assigned a type. Inputs may also be assigned a value, which is defined using the *hasValue* relationship on the input object. Procedures can specify a number of subtasks using the *hasSteps* property which links to a sequence of subtasks and/or loops. Loops are expressed using a repeat-until construct. The *repeat* property links a loop to a set of subtasks and/or loops. The until-portion of the loop is defined by the *until* property, which links to a task that should always return a Boolean value. We note that the language does not differentiate between calls to subtasks and primitive tasks. While this language is simple, it is sufficient to describe a variety of procedures. This representation can be converted to a number of formats.

Instruction Markup

Based on the above procedure language, we have devised an initial set of instructor markup that allows for the expression of complete, detailed (but not natural) tutorial instruction, and allows for the omission of instruction elements resulting in more natural statements. Although humans will prefer providing instructions in natural language, the instruction markup we describe here can be seen as the output of a natural language interpretation system (e.g., (Webber et al. 1995; Huffman and Laird 1995), or as the target of controlled English for an interface generator (e.g., as in (Fuchs, Kaljurand, and Schneider 2006)).

Tutorial instruction is divided into lessons where each lesson describes a particular procedure and is assumed to build on prior lessons. Each lesson consists of a sequence of instruction statements containing keywords that represent the markup. At the beginning of each lesson, the instructor provides a name to the lesson using the *start* keyword. Like-

wise, the instructor can provide an exact identifier to be used in subsequent lessons with the *hasId* keyword. The procedure's inputs and results are defined using the *initSetup* and *resultIs* keywords respectively. Variables are identified using the *name* keyword. They can be assigned a type using the combination of the *isa* and *type* keywords. Likewise, they can be assigned a value using the *hasValue* and *value* keywords. Using these constructs, the instructor can express constant values.

In subsequent instructions, the instructor can specify that the student should do a particular subtask using the *doThis* keyword. The name of the task to be invoked is specified using the *name* keyword. The variables acting as input are identified either using the *basedOn* or *from* keyword. The results of the task invocation are identified using the *expect* keyword. Variables can be introduced at any time during instruction and the student can be told to reassign the value of a variable to another variable using the *remember* keyword. Finally, the instructor can tell the student to repeat a number of task until another task returns true using the *repeat* and *until* keywords.

This instructor markup provides a number of scaffolding instructions that a human instructor would not always employ. First, the instructor markup may assume that the student knows the input to a particular procedure or task and thus would not make use of the *basedOn*, *from*, or *initSetup* keywords in their instruction. Second, an instructor may not explicitly tell the student what to expect from a task they are instructed to perform and thus would not use the *expect* keyword. Third, the instructor may assume that the student would know the type of the entity being used in a task from prior knowledge and thus, not always employ the *type* keyword. Finally, the student should be able to handle instructions given out of order (i.e. instruction should be able to refer to subtasks that have not yet been taught).

Examples of Instruction

We now provide two examples of instructor markup and illustrate the scaffolding that our system can eliminate. The first example, shown in Figure 2, is a description of how to setup a card game of Klondike solitaire. We assume that the student already knows about CardDeck, Hands, Deal, Decrement, Equals, and Layout. The instruction in Figure 2 mirrors the following textual natural instruction:

To setup for a game of solitaire, take a deck of cards and deal seven cards. Layout the cards horizontally on the table with the first card face up. Using one less card each time, repeat dealing and laying out the cards.

Applying the relaxations discussed in the previous section, the grayed out instructions can be eliminated from the instruction set. Additionally, the lesson for the Layout procedure, for example, can be taught after this lesson. We note that the textual instruction does not state the termination condition and yet it is shown in the markup in Figure 2. This is one kind of scaffolding that our system needs.

The second example of instruction is taken from the RoboCup soccer domain. It describes how to get open for

a ball. The instructions are shown in Figure 3 and the textual natural instruction is as follows:

Find your closest opponent. Dash away from them. Stop and move back to your previous position (e.g. cut back). If you are not open, do this again. Once your open, find the ball and face it.

By eliminating just some of the scaffolding in the instruction markup, a more natural description is approached. For instance, by eliminating the *basedOn* keyword in line 6 of the instruction, the instruction "FindClosestOpponent expect opponentLocation" resembles the instruction "Find your closest opponent".

```

1: begin lesson
2: start SetupKlondikeSolitaire hasId 8887
   // To setup for a game of solitaire,
3: resultIs name=solitaireGameSetup isa type=GameSetup
   // take a deck of cards and
4: initSetup name=deck isa type=CardDeck
   // deal seven cards.
5: name=numberOfCards hasValue 7
6: doThis name=Deal basedOn deck, numOfCards
7:   expect name=hand
8: name=hand isa type=Hand
   // Layout the cards horizontally
   // on the table with the first card face up
9: doThis name=Layout basedOn hand
   // Using one less card each time,
   // repeat dealing and laying out the cards
10: repeat
11:   doThis name=Decrement basedOn numOfCards
      expect numOfCards
12:   doThis name=Deal basedOn deck, numOfCards
      expect hand
      name=hand isa type=Hand
13:   doThis name=Layout basedOn hand
      until
14:   name=Equals basedOn numOfCards, 1
15: end lesson

```

Figure 2: Instructions for Setting up Klondike Solitaire. The typewriter font shows scaffolding that can be removed from the instruction, resulting in instruction that the current system can learn from.

Currently, our implementation of the framework, TellMe, handles the following ambiguities in instruction:

- Some missing subtask inputs and results.
- Some missing primitive types (e.g. integer).
- Some missing types on subtask results.
- Out-of-order lessons.
- Multiple procedure definitions having the same signature.

However, in order for TellMe to rectify these gaps in instruction the following scaffolding is required:

- Terms must be common across both the instruction and the student's knowledge base.

```

1: begin lesson
2: start GetOpen hasId 8888
3: repeat
  // Find your closest opponent.
4:   doThis name=GetCurrentPosition expect originalPosition
5:   name=originalPosition isa type=Position
6:   doThis name=FindClosestOpponent
     basedOn=originalPosition
     expect=opponentLocation
  // Dash away from them
7:   name=opponentLocation isa type=Position
8:   doThis name=FaceAwayFrom
     basedOn opponentLocation
9:   doThis name=Dash expect=currentPosition
10:  name=currentPosition isa type=Position
  // Stop and move back to your previous position
  (e.g. cut back).
11:  doThis name=MoveTowards
12:  basedOn originalPosition expect=currentPosition
  // If you are not open, do this again.
13: until
14:  name=Open basedOn currentPosition
  // Once your open, find the ball and face it.
15: doThis name=FindTheBall expect=ballLocation
16: name=ballLocation isa type=Position
17: doThis name=Face basedOn ballLocation
18: end lesson

```

Figure 3: Instructions for Getting Open in a RoboCup domain

- Instruction must follow a strict order within a lesson where each step directly follows from the previous step.
- All statements within instruction must be correct though they may be incomplete.

Framework Algorithms and Heuristics

We now describe the algorithms and heuristics TellMe implements in order to learn the procedures shown in Figures 2 and 3 without the highlighted scaffolding. We proceed through each of the stage of the framework. During the ingestion stage, TellMe creates a model that reflects the procedure language concepts and properties. This provides the procedure stub on which the following heuristics operate on during the elaboration stage.

Elaboration Heuristics

The first part of elaboration is the mapping of the procedure stub to the student's internal knowledge. Currently, as previously described, a simplified approach is adopted and only one heuristic is applied, which is as follows:

- If a variable is assigned a constant in the instruction, then find a consistent basic type for it. Basic types are integer, boolean, and string.

This heuristic enables, for example, the type of the numOfCards variable in Figure 2 to be inferred. After mapping, the elaboration stage then uses the following heuristics, encoded as inference rules, to hypothesize the missing

parts of the body of the expanded procedure stub generated during mapping.

- If a subtask, T , has no results, find some step, S , immediately following the call to the subtask, take S 's inputs and make them T 's results.
- If a subtask, T , has no inputs, find some step, S , immediately preceding the call to T , take S 's results and make them T 's inputs.
- If a subtask, T , has a result, R , with no type, take (arbitrarily) one of T 's inputs and make R 's type the type of that input.

Because different combinations of these heuristics may result in different procedure hypothesis, the power set of these heuristics is applied. Because there are three heuristics, a space of 32 alternative procedure hypothesis is generated. We now describe the selection algorithm used to pare down this hypothesis space.

Selection Algorithm

The selection algorithm described below is symbolic execution algorithm that provides a trace of its execution to enable the comparison of various procedure hypothesis. The selection algorithm works over the procedure language described above and the student's internal knowledge base. The algorithm relies on the following definitions.

- A signature, sig , is defined as a tuple containing a name, s_{name} , a list of arguments containing name, type pairs, $\{(a_{name}, a_{type}), \dots\}$ and a result pair (r_{name}, r_{type}) :
 $sig = \langle s_{name}, \{(a_{name}, a_{type}), \dots\}, (r_{name}, r_{type}) \rangle$.
- A procedure, p , is defined as a tuple containing a signature, sig , and a body that consists of a list of signatures, SIG :
 $p = \langle sig, SIG \rangle$
- A knowledge base, KB , is both a map of signatures to procedures and a map of name, type pairs to values:
 $KB[sig] \rightarrow SIG, KB[(x_{name}, x_{type})] \rightarrow v$
- A state, S , is a map from a name, type pair to a value, v :
 $S[(x_{name}, x_{type})] \rightarrow v$
- A table, $PrimitiveTasks$ that maps a signature name $name$, and a list of type, value pairs, A , to a skolem result value, r :
 $r \leftarrow PrimitiveTasks[name, A]$
- A trace, t , is a tree where the tree nodes are named by tuples containing a signature, state, and boolean value, where the boolean value describes whether the task defined by the signature was successful in executing upon the state. The branches in the tree describe the branches of execution.
- A trace node, tn , is defined as reference to a node in a trace.
- A result, r , is a tuple containing a trace and a constant, g , denoting the whether the symbolic execution was a success or failure.

```

SELECT(KB, p)
1  create a blank state, S
2  for each argument, (aname, atype) in p.sig
3      do create a skolem value, v, of atype
4          ▷ Put the skolem value in the state
5           $v \rightarrow S[(a_{name}, a_{type})]$ 
6  create a new trace, t
7  get the root node, tn of trace, t
8  ▷ Begin the symbolic execution
9  return PROCWALK(KB, S, p, tn)

```

Figure 4: Pseudo-code for starting the selection algorithm

- We use a dot accessor notation to denote accessing fields in a tuple. Hence, *sig.sname* would “retrieve” the name for the particular signature.

The selection algorithm begins in the SELECT method, which takes a procedure and generates skolems (i.e. symbols) for each of the inputs of the procedure as defined by its signature, *p.sig* and adds these to the state. SELECT then calls the PROCWALK method.

The PROCWALK method symbolically executes each of the steps defined in a procedure by either executing all the procedure definitions that correspond to the step’s signature or, if there are no procedure definitions corresponding to the signature, checking whether there is a primitive task that can perform the step. Thus, the algorithm supports checking multiple ways of performing the same task. For example, there may be two procedures for moveTowards in the RoboCup example, the algorithm would check whether both are consistent with the GetOpen procedure hypothesis. Note, that a step fails (and thus the procedure fails) only if all of possible procedure definitions also fail as shown starting on PROCWALK line 26 or when no procedure definitions exist, there is no primitive task for that step. A procedure will fail if one of its steps fails or the state does not contain the necessary symbols for the procedure to execute as defined by the procedure’s signature.

The PROCWALK method checks whether primitive tasks exist using the CHECK method, which looks up the provided signature in a table, which maps the signature to a symbolic value.

For conciseness, the algorithm presented here does not treat loops. However, our implementation does symbolically execute loops. As we are focused on consistency, the implementation does not execute the loop iteratively until its condition has been met, but instead treats the loop body as if it were a procedure body and executes it as such. The implementation also tests to ensure that the loop’s condition is also executable.

Once finished, the selection algorithm returns a result consisting of a trace and a result constant. If the result constant is FAIL then the procedure hypothesis is discarded. However, if the constant is a success, the hypothesis is retained as a plausible procedure hypothesis. To reduce the number of plausible procedure hypothesis even further, we make use of the trace associated with each hypothesis. It is often the case that only some heuristics used in the elab-

```

PROCWALK(KB, S, p, tn)
1  create a new tree node,  $tn_p = (p.sig, S, false)$ 
2  add  $tn_p$  as a child of tn.
3  ▷ Check to see if the state contains the arguments
4  ▷ necessary for the given procedure,
5  ▷ if the state does not have an appropriate value
6  ▷ try and load a default value from the knowledge base.
7  ▷ If that does not work, fail.
8  for each argument, (aname, atype) in p.sig
9      do if S does not contain (aname, atype)
10         then
11             if  $v \leftarrow KB[(a_{name}, a_{type})]$  is not NIL
12                 then  $v \rightarrow S[(a_{name}, a_{type})]$ 
13             else return ( $tn_p$ , FAIL)
14  ▷ Go through the body of the procedure
15  ▷ (i.e. the list of signatures referenced by the procedure).
16  ▷ Either the KB has a procedure definition that ,
17  ▷ matches the signature or
18  ▷ the procedure is a primitive task, which can be checked.
19  for each sig ∈ p.SIG
20      do  $\{p_1, \dots p_n\} \leftarrow KB[sig]$ 
21      if  $\{p_1, \dots p_n\}$  is NIL ▷ i.e. not in the KB
22          then
23              if CHECK(sig, S,  $tn_p$ ) is NIL
24                  then return ( $tn_p$ , FAIL)
25          else
26              oneSuccessfulProc = false
27              for each  $p_x \in \{p_1, \dots p_n\}$ 
28                  do
29                       $r = PROCWALK(KB, S, p_x, tn_p)$ 
30                      if r.g is SUCCESS
31                          then oneSuccessfulProc = true
32              if oneSuccessfulProc is false
33                  then return ( $tn_p$ , FAIL)
34   $tn_p = (p.sig, S, true)$ 
35  return ( $tn_p$ , SUCCESS)

```

Figure 5: Pseudo-code for walking through a procedure decomposition

```

CHECK(sig, S, tn)
1  create a new tree node,  $tn_c = (sig, S, false)$ 
2  add  $tn_c$  as a child of tn.
3  ▷ Check to see if a primitive task is
4  ▷ defined for the particular signature but without using
5  ▷ any argument names.
6  A is a list of pairs of argument types and values
7  for each argument, (aname, atype) in sig
8      do  $v \leftarrow S[(a_{name}, a_{type})]$ 
9      add the pair (atype, v) to A
10  $r \leftarrow PrimitiveTasks[signame, A]$ 
11 ▷ If a skolem result is returned by
12 ▷ the table lookup update the state.
13 if r is not NIL
14     then
15          $tn_c = (sig, S, true)$ 
16          $r \rightarrow S[sig.((r_{name}, r_{type}))]$ 
17 return r

```

Figure 6: Pseudo-code for looking up primitive tasks

oration stage actually apply to a particular set of tutorial instruction. Hence, when applying the power set of these heuristics, identical procedure hypotheses are created by the application of different sets of heuristics. To detect whether a procedure hypotheses is identical, their traces can be compared. For example, the SetupKlondikeSolitaire instructions result in 8 plausible procedure hypotheses, but after comparing traces, it was determined that there was only one true hypothesis. As our algorithms support increasingly ambiguous instruction, the number of plausible hypothesis will undoubtedly increase. Our aim is to leverage these traces to rank and select the best hypotheses.

Implementation

TellMe is implemented as a Java application. To enable integration with preexisting knowledge sources and to leverage existing reasoning systems, we chose the W3C recommended ontology language OWL to represent procedures and their properties. All assertions produced by the system result in triples. The heuristics for inferring missing procedure knowledge are implemented as Jena rules, which are applied using the Jena Semantic Web Framework's (jena.sourceforge.net) built-in forward chaining inference mechanism.

Results

TellMe can output a variety of formats. Here, the output is formatted for the planner SHOP2 (Nau et al. 2003). Figures 7 and 8 show excerpts of the procedure representations learned from the instructions shown in Figures 2 and 3 (type-writer text scaffolding was omitted for learning). Primitive tasks are denoted by a “!”. Subtasks (i.e. methods) are not prefixed by “!”. Variables are denoted by “?”. Loops in our procedure language are translated into recursive calls. Currently, only procedure definitions are output and not primitive tasks. Each excerpt shows the definition of each procedure as well as particular steps the procedure calls. The portions of the procedure language highlighted in bold are inferred in the elaboration stage. For example, in the description of the Deal step shown in Figure 7, the result of the step was inferred. Likewise, in Figure 8, the input to FaceAwayFrom is inferred. Furthermore, Figure 7 shows that two different procedure definitions for Layout were produced for the same signature required by SetupKlondikeSolitaire.

Conclusion

We have presented an approach to learn procedures from human tutorial instruction. This approach deals with a number of deficiencies in instruction including undefined procedure inputs and outputs, missing typing information, and out-of-order of lessons. However, to cope with these deficiencies, the system still requires scaffolding, namely, the use of common terminology between the student and teacher, well ordered instruction, and correct instruction. In future work, we aim to remove this scaffolding through the addition of new elaboration heuristics, a more complex mapping approach, and selection heuristics based on the traces produced during

<pre>(:method :head (SetupKlondikeSolitaire ?deck) :precondition () :subtasks (:ordered (!Deal 7 ?deck) (Layout ?hand) (Loop0)))</pre>	<pre>(:method :head (Loop0) :precondition (!Equals ?numOfCards 1) :subtasks (:ordered (!Decrement ?numOfCards) (!Deal ?numOfCards ?deck) (Layout ?hand) (Loop0)))</pre>
<pre>(:method :head (Layout ?hand) :precondition () :subtasks (:ordered (!DrawOne ?hand)))</pre>	<pre>(:method :head (Layout ?hand) :precondition () :subtasks (:ordered (!GetTopCard ?hand) (!TurnFaceUp ?cardInHand) (!Place ?startLocation ?cardInHand) (!GetTopCard ?hand) (!PlaceNextTo ?lastPlacedCard ?cardInHand) (!GetLocationOfCard ?cardInHand) (Loop1)))</pre>

Figure 7: Excerpt of the learned domain for SetupKlondikeSolitaire

<pre>(:method :head (GetOpen) :precondition () :subtasks (:ordered (Loop0) (FindTheBall) (!Face ?ballLocation)))</pre>	<pre>(:method :head (Loop0) :precondition (!Open ?currentPosition) :subtasks (:ordered (!GetCurrentPosition) (!FindClosestOpponent ?originalPosition) (!FaceAwayFrom ?opponentLocation) (!Dash) (!MoveTowards ?originalPosition) (Loop0)))</pre>
---	--

Figure 8: Excerpt of the learned domain for GetOpen

symbolic execution. Furthermore, we aim to use existing procedures to bootstrap the internal knowledge of the student before tutorial instruction.

This paper explored the design of electronic students that can be extended over time with new capabilities that enable them to learn procedures from increasingly more natural instruction as quantified by the removal of scaffolding statements. Concretely, we presented a framework for dealing with tutorial instruction, a markup language that expresses instruction details piecemeal so they can be removed and a set of heuristics for inferring missing procedure knowledge. Additionally, we presented a symbolic execution algorithm for checking the consistency of procedure hypothesis. Finally, we described an implementation of the electronic student framework that demonstrates in two domains that it can learn procedures from incomplete tutorial instruction.

References

- Allen, J.; Chambers, N.; Ferguson, G.; Galescu, L.; Jung, H.; Swift, M.; and Taysom, W. 2007. PLOW: A Collaborative Task Learning Agent. In *Proceedings of the 22nd AAAI Conference on Artificial Intelligence (AAAI-07)*.
- Blythe, J.; Kim, J.; Ramachandran, S.; and Gil, Y. 2001. An Integrated Environment for Knowledge Acquisition. In *Proceedings of the International Conference on Intelligent User Interfaces (IUI-2001)*.
- Britt, M., and Larson, A. 2003. Construction of argument representations during on-line reading. *Journal of Memory and Language* 48(4).
- Clark, P.; Thompson, J.; Barker, K.; Porter, B.; Chaudhri, V.; Rodriguez, A.; Thom  r  , J.; Mishra, S.; Gil, Y.; Hayes, P.; and Reichherzer, T. 2001. Knowledge entry as the graphical assembly of components. In *K-CAP '01: Proceedings of the 1st international conference on Knowledge capture*, 22–29.
- Dixon, P.; Faries, J.; and Gabrys, G. 1988. The role of explicit action statements in understanding and using written directions. *Journal of Memory and Language* 27(6):649–667.
- Etzioni, O. 1993. Acquiring search-control knowledge via static analysis. *Artificial Intelligence* 62(2):255–301.
- Fuchs, N. E.; Kaljurand, K.; and Schneider, G. 2006. Attempto Controlled English Meets the Challenges of Knowledge Representation, Reasoning, Interoperability and User Interfaces. In *FLAIRS 2006*.
- Gil, Y., and Melz, E. 1996. Explicit representations of problem-solving strategies to support knowledge acquisition. In *AAAI/IAAI, Vol. 1*, 469–476.
- Huffman, S. B., and Laird, J. E. 1995. Flexibly instructable agents. *Journal of Artificial Intelligence Research* 3:271–324.
- Kaelbling, L. P.; Littman, M. L.; and Moore, A. P. 1996. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research* 4:237–285.
- Kim, J., and Gil, Y. 2001. Knowledge analysis on process models. In *IJCAI*, 935–942.
- King, J. C. 1976. Symbolic execution and program testing. *Communications of the ACM* 19(7):385–394.
- Lau, T.; Wolfman, S. A.; Domingos, P.; and Weld, D. S. 2003. Programming by demonstration using version space algebra. *Machine Learning* 53(1-2):111–156.
- Lieberman, H. 1994. A user interface for knowledge acquisition from video. In *AAAI '94: Proceedings of the Twelfth National Conference on Artificial intelligence (vol. 1)*, 527–534.
- Maclin, R., and Shavlik, J. W. 1996. Creating advice-taking reinforcement learners. *Machine Learning* 22(1-3):251–281.
- Mahling, D. E., and Croft, W. B. 1989. Relating human knowledge of tasks to the requirements of plan libraries. *International Journal of Man-Machine Studies* 31(1):61–97.
- Nau, D. S.; Au, T. C.; Ilghami, O.; Kuter, U.; Murdock, J. W.; Wu, D.; and Yaman, F. 2003. SHOP2: An HTN Planning System. *Journal of Artificial Intelligence Research* 20:379–404.
- Reiser, B. J. 2004. Scaffolding Complex Learning: The Mechanisms of Structuring and Problematising Student Work. *Journal of the Learning Sciences* 13(3):273–304.
- Steehouder, M.; Karreman, J.; and Ummelen, N. 2000. Making sense of step-by-step procedures. In *Proceedings of the 18th annual ACM International Conference on Computer Documentation*, 463–475.
- Swartout, W.; Paris, C.; and Moore, J. 1991. Explanations in knowledge systems: Design for explainable expert systems. *IEEE Expert* 06(3):58–64.
- Webber, B.; Badler, N.; Eugenio, B. D.; Geib, C.; Levison, L.; and Moore, M. 1995. Instructions, intentions and expectations. *Artificial Intelligence* 73(1-2):253–269.
- Winner, E., and Veloso, M. 2002. Analyzing plans with conditional effects. In *Proceedings of the Sixth International Conference on Artificial Intelligence Planning Systems*.
- Yang, Q.; Wu, K.; and Jiang, Y. 2007. Learning action models from plan examples using weighted MAX-SAT. *Artificial Intelligence* 171(2-3):107–143.