Generating Natural Language Expectations from a Reactive Execution System

Charles E. Martin and R. James Firby
Department of Computer Science
University of Chicago
1100 East 58th Street
Chicago, IL 60637
martin@cs.uchicago.edu

Objectives

We are developing an integrated system for planning, language understanding, and learning through *advice-taking* (Martin and Firby, 1991). Our work brings together two reasonably mature pieces of research: the Reactive Action Package (RAP) execution system of Firby (1989), and the Direct Memory Access Parser (DMAP) of Martin (1991). Specifically, we are reformulating the RAP plan representation and execution algorithm in terms of the semantic network architecture used by the DMAP system. The result is a system with a uniform knowledge representation and processing algorithm that can both act in the world and process natural language. Furthermore, acting and understanding use a single process so the system can change its actions based on language inputs. That forms a basis for learning new plans and actions by being told.

Although "learning by being told in natural language" might strike one as a natural area of research in machine learning, a look through the proceedings of the last three Machine Learning Workshops (ML 1988, 1989, and 1990) and *Readings in Machine Learning* (Shavlik and Dietterich, 1990) will demonstrate that there is in fact *no* current machine learning research in this area. Most research on "advice-taking" has concentrated on the operationalization of *already-interpreted* input; an excellent example is Mostow's (1983) research. Broadly construed, this notion could be expanded to include *any* system that attempts to improve its decision-making data structures using an oracle, guided examples, etc. This is not our goal.

Instead, we have two complementary hypotheses:

1. The natural language understanding process can be greatly simplified by embedding the understander in another task-oriented system, where "embedding" refers to the use of *identical* data structures and a *single* algorithm for both the task and the natural language understanding process.
2. The performance of a task-oriented system can be extended by learning specific local plan modifications, derived by the overall system on the basis of high-level "hints" supplied in natural language.

In our research, the reactive execution system is the task-oriented system in which we embed the natural language understander, and the human expert provides high-level "hints" that are the basis for extending the reactive execution system's performance.

Background

The DMAP system is designed to function within a large body of existing knowledge. Rather than determine the meaning of a text, DMAP uses the text to recognize relevant existing knowledge structures and then modify them to create specific new instances in memory that represent what is unique about the given text. New instances remain in memory to aid in the interpretation of future texts.

The RAP system is designed to carry out vague goals by waiting until execution time and then expanding the goal into primitive actions based on the details of the situation actually encountered. Possible plan expansions are stored in a hierarchical library of methods indexed by the goal they satisfy. The RAP interpreter selects a method from the library based on existing context and either instantiates the method as a new set of goals to achieve or executes the method directly if it consists of a primitive action. When a method completes (or fails) the interpreter checks to see whether the goal was actually satisfied in the world and, if not, selects and instantiates another method.

In bringing these two systems together we are directed by two "implementation" constraints:

1. There must be a uniform representation for all knowledge in the system, regardless of whether it is used for natural language understanding or planning and execution.
2. There must be a uniform algorithm to operate over the system's knowledge, regardless of whether the system is engaged in natural language understanding or planning and execution.

The first constraint is hardly new to devotees of KRL and its many descendants, though the constraint is honored more often in the breach than in the observance. The second constraint is more radical; in ef-

fect, it states that language understanding and planning and execution must be the *same* process. This is our hypothesis, with the caveat that differences between acting and understanding do exist, and they will manifest themselves in the system's *representation*.

The RAP model of execution provides a single, coherent representation for planning and execution knowledge with a simple interpretation semantics that does not require the maintainence of complex dependency structures. The DMAP model of language understanding relies on expectations about language that arise as a result of moving around in a semantic net. To combine the two, we have reformulated the RAP knowledge representation in terms of a DMAP-style semantic network that preserves the execution semantics when traversed in accord with the DMAP algorithm. The result is a single representation and processing algorithm for both acting on goals and processing natural language.

We have implemented our initial system in the robot-truck simulator domain described in Firby and Hanks (1987) and used by the RAP system as described in Firby (1989). We are currently building initial representations for the same RAPs used in that work and we anticipate working in this domain for some time to create a large DMAP system and refine our ideas. Ultimately, we intend to demonstrate these ideas on the robotic platform being built at the University of Chicago.

Overview of the system

In keeping with the DMAP approach, the combined system expresses interpretation, planning, and execution as *recognition tasks*. Intuitively, the idea is that the system must have something similar to what it supposed to be doing already in its knowledge base. For natural language understanding, the system must already know a concept similar to that being communicated for the text to be understood correctly. For task execution, the system must already know methods that will work for the current goal in common circumstances. Naturally, new elements must be added to account for the differences between what exists and what is newly expressed or executed. For interpretation, these elements represent the content of the communication; for execution, these elements represent the intention and record of action.

Representation

Representation is in the form of a semantic network, organized by abstraction and labelled packaging relationships. This representation was chosen primarily for the efficiency of the algorithm. Extensions to this basic framework include the specification of constraints on the packaging relationships of sub-structures to ensure that variable bindings make sense. There is also a system of *indices* used by the processing algorithm as a means of maintaining ordering relationships between concepts.

For example, the following memory units represent a decomposition of the task of picking an object up with a robot arm. There two subtasks: moving the arm to the location of the object and grasping the object with the arm. Constraints (the non **isa:** and **index:** forms) assure that variable binding is handled correctly. The index form indicates that in order to *recognize* (or satisfy)[1] this unit, the two sub-units, **task1** and **task2** must be recognized in the given sequence, followed by a recognition of the success condition for the task. Some of this information (such as the success condition) is inherited from its abstraction.

```
(define-unit task
  (success state))

(define-unit arm-pickup
  (isa: task)
  (arm ?arm)
  (object ?object)
  (success holding (arm ?arm) (object ?obj)))

(define-unit arm-pickup-simple
  (isa: arm-pickup)
  (task1 arm-move  (arm ?arm) (object ?obj))
  (task2 arm-grasp (arm ?arm) (object ?obj))
  (index: (task1) (task2) (success)))
```

The following is a more complex task to pick up an object, which involves first picking up the appropriate tool for the object and then the object itself. Note the use of the **instr** constraint to assure that the **?tool** is appropriate for the **?obj**. Notice also that this unit and the **arm-pickup-simple** unit both inherit the same success condition from **arm-pickup**.

```
(define-unit arm-pickup-w/tool
  (isa: arm-pickup)
  (instr instrument (tool ?tool) (object ?obj))
  (task1 arm-pickup-simple (arm ?arm) (object ?tool))
  (task2 arm-pickup-simple (arm ?arm) (object ?obj))
  (index: (task1) (task2) (success)))

(define-unit instrument
  (tool tool)
  (object object))
```

Algorithm

The basic DMAP algorithm is quite simple:

RECOGNIZE (*concept*):

[1] Processing in the DMAP memory is driven by memory unit indices. In language understanding, indices represent expectections for particular words, phrases, and abstract concepts. Processing thus consists of recognizing words, phrases, and concepts as they arrive from a text. In extending DMAP to include execution, indices represent intentions as well as expectations. Processing thus consists of both recognizing expectations and satisfying intentions. The algorithm remains the same, but the intuitive nomonclature shifts a little. We will use the words recognize and satisfy interchangably while discussing the processing algorithm.

If *concept* is a primitive operation
then apply it and process pending sensor inputs;
else gather *indices* to recognize *concept*,
for each *index* in *indices*,
for each *element* of the *index*,
call RECOGNIZE (*element*).

Primitive operations can be either sensor expectations (such as measurements or words) the robot must wait for or effector actions that the robot can execute. Effector actions include movements and active perception. When a sensor expectation is part of a memory unit index, recognition of that unit is blocked until data matching the expectation is generated by a sensor. When an effector action is part of an index, it is executed immediately and that execution may generate sensor data.

Successful execution of an effector operation or the satisfaction of a sensor expectation both result in the "recognition" of the corresponding <*element*> of the <*index*>; if all such <*elements*> have been recognized, the corresponding <*concept*> is recognized. Recognition of a unit causes all other units with the recognized unit as an index element to *specialized* the next element in their index. Specializing a unit involves moving down the `isa:` hierarchy to find the most specific memory unit that satisfies existing binding constraints. Any newly discovered specializations are *instantiated* by creating new versions of them in memory with variables bound according to the constraints. Units without indices are always specialized when instantiated and that is what moves activation of `arm-pickup` to either `arm-pickup-simple` or `arm-pickup-w/tool` depending on the binding of `?obj`.

When "gathering indices," all indices associated with the <*concept*> and any of its abstractions are collected. The system attempts to recognize all such indices through "recursive" calls to RECOGNIZE. If a complete unit index is recognized at any level of abstraction, that unit and all of its subunits are recognized together. That way, goals and concepts can be recognized when they occur even if no specific method or expectation set produces the required result.

For more details of the algorithm, see Martin (1990). For more background on this research, see Martin and Firby (1991).

Execution Problems and Advice

Consider the following situation: The robot has task (1) "pick up a rock." It decomposes this task into two subtasks: (1A) "release anything currently held in the arm," and (1B) "pick up the rock with the arm." The latter task is in turn decomposed into the two subtasks: (1B1) "move the arm to the rock," and (1B2) "grasp the rock." Unfortunately, the grasping action (1B2) fails when executed because the arm is not well constructed for picking up rocks.

A friendly human wanders by and sees the robot's plight. The human instantly sees the problem, and knows that the grasping action would have succeeded if the robot had previously attached the shovel tool to its arm. There is precedent for this in the robot's memory; had the original task been to pick up a fuel-drum, the robot would have decomposed task (2) "pick up a fuel drum" into (2A) "release anything currently held in the arm," and (2B) "pick up the fuel-drum with the arm." The decomposition of (2B) would have been into three tasks: (2B1) "attach refueling tool," (2B2) "move the arm to the fuel-drum," and (2B3) "grasp the fuel-drum."

In other words, the general plan for (1B) and (2B), "pick up <*object*> with the arm" is able to discriminate between task decompositions on the basis of the <*object*>. This discrimination, of course, is represented in the vocabulary of plan descriptions. From the point of view of the robot, what is required to fix the rock problem is for someone to sit down and reprogram the plan library so that the general plan for (1B) and (2B) knows about rocks (and their associated gripper) as well as fuel-drums.

Unfortunately, the human who wanders by is not a robot programmer! (Perhaps the human builds robot gripper-tools.) From the point of view of the human, what is required is for the robot to somehow comprehend that this handy shovel tool should be used to pick up rocks in the same way that the refueling tool is used to pick up fuel-drums *without* having to have it hand-coded into the plan library. What the human would like to do, of course, is to simply say, "use the shovel."

Our goal is a system that lets the human do just that.

The Context of Advice-Taking

There are a number of extra-sentential elements that are important to understanding "use the shovel" as useful advice. Chief among these is the fact that the robot is actively engaged in a specific task which it is *failing* to accomplish. This is a crucial piece of information; imagine the interpretation of "use the shovel" when the robot is not having a problem: "for what?" might well be the most appropriate response. However, in this case, the clear implication is that the shovel will be of use for the current task.

This is essentially a statement of the advice-taking problem our system is designed to address. The remainder of the paper describes the way we have approached the problem of establishing the situational context for the interpretation of an utterance, involving as it does tasks of both interpretation and execution. The context we will discuss is the internal state of the robot and not the external state of the world, except for knowledge that a current action is not succeeding.

Execution Failures

In our example, an execution failure occurs when the robot attempts to grasp the rock without the appropriate tool.

```
(define-unit primitive
  (isa: task)
  (primop $primitive)
  (index: (primop) :ok (success)))

(define-unit arm-grasp
  (isa: task)
  (arm ?arm)
  (object ?object)
  (success holding    (arm ?arm) (object ?object))
  (primop $arm-grasp (arm ?arm) (object ?object)))
```

The goal of picking up the rock results in an attempt to recognize **arm-grasp** which results in an attempt to recognize, in sequence, the primitive **$arm-grasp**, the **:ok** message from the effectors stating that the primitive has been executed correctly, and the *success* condition of the task. If the robot hasn't picked up the appropriate gripper, the sequence of activity will approximate the following:

1. The attempt to recognize **$arm-grasp** succeeds, since an *attempt* at a primitive action always succeeds.
2. The attempt to recognize **:ok** fails, since the *result* from the primitive action is a notification that the grasp failed . In this case, the failure occurs because robot did not have the proper tools for success.

All recognition attempts which fail—whether from an attempt to recognize an execution sequence or a natural language utterance—result in the automatic recognition of a **failure** structure. This will result in the specialization of concepts that have the appropriate failure as an index and the eventual instantiation of a repair memory unit.

The **failure** hierarchy parallels every hierarchy which has indices; that is to say, virtually every hierarchy in the system. Every **failure** has a *source*, which is the structure whose recognition failed. Moreover, because each recognition task is the result of a "recursive call" to RECOGNIZE, the failure of one such task implies the failure of its parent and so on, up the chain of calls. Thus, the failure of the arm-grasp operation results in the recognition—and hence construction—of multiple failure structures.
Assume that the specific instances of the active pickup units involved in our failure are the following:

```
(instance arm-pickup-simple-1
  (isa: arm-pickup-simple)
  (arm arm-2)
  (object rock-3)
  (task1 arm-move-4)
  (task2 arm-grasp-5))

(instance arm-grasp-5
  (isa: arm-grasp)
  (arm arm-2)
  (object rock-3))
```

In this case, the failures that are generated and recognized are:

```
(instance failure-6
  (isa: failure)
  (source arm-grasp-5))

(instance failure-7
  (isa: failure)
  (source arm-pickup-simple-1))
```

Generating Expectations from Failures

The recognition of these failure structures may, as with any successful recognition, cause the refinement and activation of further structures. The only structures that make reference to **failure**s in their indices are those in the **repair** hierarchy. When a **failure** structure is recognized, the DMAP system attempts to refine the corresponding **repair** structures. One such candidate for refinement is **repair-task/advice**, given below:

```
(define-unit repair
  (failure failure))

(define-unit repair-task
  (isa: repair)
  (failure failure (source task)))

(define-unit repair-task/advice
  (isa: repair-task)
  (failure failure (source ?old-task (success ?success)
  (advice  mtrans (info ?new-task (success ?success)))
  (attempt ?new-task)
  (index: (failure) (advice) (attempt)))
```

The index to this structure implies that three recognition tasks take place. The first, recognition of the failure which has happened already, results in creating a context that includes a constraint defining a *communication event*, signified here by **mtrans**, in which the content of the communication is a **task** that shares the success condition of the original task—that is, a piece of advice.

This second element of the index is not an action that can be undertaken by the robot, but instead represents a prediction that natural language input might be forthcoming and that it will satisfy the advice constraint. In other words, upon the robot's failure to execute a task, it will always expect that it may receive a communication of a task sharing the success condition of the task that failed. This communication will be a piece of advice on how to achieve the success condition another way.

Interpretating Advice

In our example, the advice "use a shovel" is meant to convey to the robot that the shovel is a tool which the robot should pick up before trying to pick up rocks. Note that the "tool-nature" of the shovel is not explicit

in the utterance, nor is the fact that it is to be used for rocks. The expectations set up by refining failures to repairs sets up the context necessary for the robot to understand that the phrase, "use a shovel" implies this advice. Consider the following memory structures:

```
(define-unit mtrans
  (info mobject))

(define-unit mtrans-task-w/tool
  (isa: mtrans)
  (info task-w/tool (instr instrument (tool ?tool)))
  (index: use the (info instr tool)))

(define-unit task-w/tool
  (isa: task)
  (instr instrument))
```

The task to recognize the **(advice)** of the earlier repair structure results in instantiation of (among other things) **mtrans-task-w/tool** and an attempt to recognize its index. The nature of this index and memory unit is that the word "shovel" is interpreted as a specification of the **?tool** in the task structure.

Having interpreted "shovel" as a tool, recognition of the **mtrans-task-w/tool** is complete which completes recognition of the **advice** element in the repair structure index. The next element is **attempt** which must be instantiated as a specialized memory unit with the following constraints:

1. It must be a **task**.
2. Its *success* condition must be that of the original task.
3. It must have an *instr* role whose *tool* is a **shovel**.

These conditions are satisfied by the **arm-pickup-w/tool** structure, so the recognition algorithm creates a specific instance of this structure.

Using Advice

Now that a task has been instantiated for the **attempt** element of the index associated with **repair-task/advice**, the next step is to recognize it. This task is an instance of **arm-pickup-w/tool**, and will require picking up the tool (the **shovel**) first, followed by picking up the object. Presumeably, this attempt will succeed.

The general recognition algorithm described above operates by creating specific instances of recognized memory structures. Many of these are too specific to be of later use to the system; for example, those that deal with picking up **rock-38**. As a consequence of instantiating these specific structures, however, more general concepts such as the fact that **shovels** are useful **instruments** for picking up **rocks** are also created.

At this point, not only has the helpful human's advice caused the robot to change to a more successful subtask, the process of interpreting "use a shovel" has instantiated an **instrument** relation in memory which can be used *anytime* in the future. Specifically, the system has learned that shovels are used for picking up rocks.

(Note that too-specific memory structures might as well be garbage-collected, except that determining which concepts are too specific is not a trivial task. A consequence of the recognition algorithm is that it will always use the most specific concept that is relevant to its inputs, but it is difficult to determine *in advance* what that level will be. Since performance does not degrade if these extra structures remain in memory, no attempt is made to remove them.)

References

Firby, J. 1989. *Adaptive Execution in Complex Dynamic Worlds.* Ph.D. dissertation, Yale University.

Firby, J. and Hanks, S. 1987. *The simulator manual.* Technical report YALEU/CSD/RR #563, Computer Science Department, Yale University.

Laird, J., Rosenbloom, P., and Newell, A. 1986. Chunking in Soar: The Anatomy of a General Learning Mechanism. *Machine Learning*, 1:11–46. Also collected in Shavlik and Dietterich (1990).

Martin, C. 1990. *Direct Memory Access Parsing.* Ph.D. dissertation, Yale University.

ML 1988. *Proceedings of the Fifth International Conference on Machine Learning*, Ann Arbor, MI.

ML 1989. *Proceedings of the Sixth International Conference on Machine Learning*, Ithaca, NY.

ML 1990. *Proceedings of the Fifth International Conference on Machine Learning*, Austin, TX.

Mostow, J. 1983. A problem-solver for making advice operational. *Proceedings of AAAI-83*, Washington, D.C., pages 279–283.

Shavlik, J. and Dietterich, T. 1990. *Readings in Machine Learning*, Morgan Kaufmann, San Mateo, CA.