Real-time Natural Language Generation in NL-Soar

Robert Rubinoff
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15123
rubinoff@cs.cmu.edu

Jill Fain Lehman
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15123
jef@cs.cmu.edu

Abstract

NL-Soar is a computer system that performs language comprehension and generation within the framework of the Soar architecture [New90]. NL-Soar provides language capabilities for systems working in a real-time environment. Responding in real time to changing situations requires a flexible way to shift control between language and task operations. To provide this flexibility, NL-Soar organizes generation as a sequence of incremental steps that can be interleaved with task actions as the situation requires. This capability has been demonstrated via the integration of NL-Soar with two different independently-developed Soar-based systems.

1 Real-time generation and NL-Soar

NL-Soar is a language comprehension and generation facility designed to provide integrated real-time[1] language capabilities for other systems built within the Soar architecture [New90]. In particular, this requires integrating NL-Soar's generation subsystem with other task(s) that Soar is performing.[2] One possible way to achieve this integration would be to use generation as a kind of "back end" which other task(s) can call as a subroutine whenever they need to say something. This approach is widely used in applications such as database query systems or expert systems, where the main system invokes the generator to express the answer to a query or to explain some aspect of its reasoning or conclusions.

In applications that need to provide real-time behavior, though, this "subroutine" approach is problematic. There is no way for the task to interrupt generation in order to handle some other (perhaps urgent) work. In addition, if generation is an unbounded process, it may proceed to complete an utterance that may have become unnecessary or even harmful because of changes in the situation; the task has no way to modify what it wants to say once generation has been invoked. While the task could of course simply stop NL-Soar in either of these cases, there is no way to guarantee that generation will be interrupted in a state from which it can recover if reinvoked later.

Furthermore, the problem isn't simply one of the speed of generation; using faster processors to run Soar won't eliminate the difficulties. It might seem that we could simply assume NL-Soar can run fast enough to finish constructing an utterance before the task has time to do anything else, this is not the case. First, generation is potentially unbounded; no matter how fast a computer is used, there will still be occasions when NL-Soar takes longer than the task can afford to wait. More significantly, this assumes that NL-Soar can absorb all the speedup; this is not reasonable. If we have faster processors, we want *all* the tasks to share the speedup equally; a faster NL-Soar will be invoked by a task that can respond more quickly as well, and will thus want to interrupt NL-Soar more quickly.

The underlying difficulty with the subroutine approach is that generation can take an unbounded amount of time; in a real-time situation, we need to guarantee that generation can't prevent the task from responding promptly to changes in the situation. Generation must be incremental and interruptible. NL-Soar accomplishes this by dividing generation into small steps that can be interleaved with task steps. In cases where the small steps can't be directly carried out and require more complex computation, the sub-steps are designed so that interruptions leave the system in a clean state (although some work may be lost and need to be redone). This allows NL-Soar to operate without limiting the task's ability to respond to things that happen during generation, and vice versa.

2 A Brief Introduction to Soar[3]

Soar is an architecture for building cognitive models and AI systems that has been applied to a wide range of problems [RLN93]. Soar carries out a task by applying a sequence of operators to the state of a problem-space until it reaches a state that solves the goal Soar is working on. When Soar is unable to directly carry out some step (e.g. selecting or applying an operator or selecting a problem space), it creates a subgoal to resolve the impasse that is preventing it from proceeding. In response to this subgoal, Soar selects an appropriate problem

[1] NL-Soar is being used in applications that perform in both simulated and actual real-time environments.

[2] Similar issues arise in NL-Soar's language comprehension subsystem, which is not described here; see [LLN91, Lew93] for a discussion of this part of NL-Soar.

[3] For a more detailed description of Soar than is possible here, see [LNR87] or [New90].

space, sets up the initial (sub-)state, and proceeds to apply operators until it determines a way to resolve the initial impasse. This process is recursive; processing in a sub-goal can itself hit an impasse leading to a further sub-goal, and so on. The elements of the Soar architecture are shown in Figure 1; numbers in the rest of this section refer to parts of the figure.

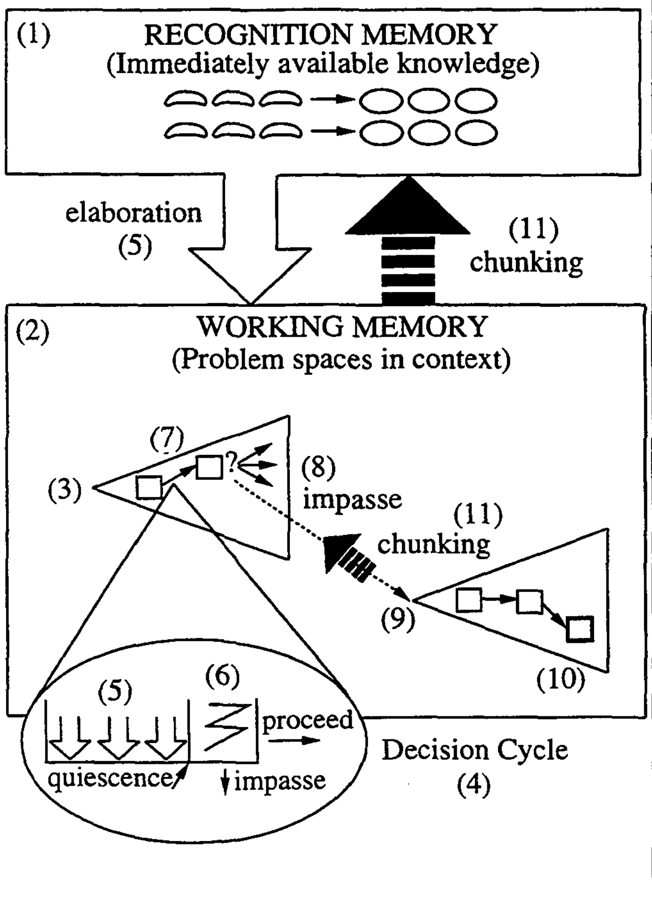

Figure 1: The Soar Architecture

Soar has two different memories: a short-term or working memory (2) and a long-term production or recognition memory (1). Working memory consists of a set of states, one state per active problem-space (3) and (9); each state has a set of attribute-value pairs. The values can be simple constants or can themselves have attached sets of attribute-value pairs; thus working memory is composed of trees (or networks) of attributes and values rooted at the states.

The production memory stores the knowledge Soar uses to carry out its processing. It contains productions whose left-hand-sides test for the presence and/or absence of structures in working memory. Production right-hand-sides indicate attributes and values to add to or remove from working memory. The right-hand-side can also contain proposals for new problem-spaces, initial states, and operators for Soar to select. Note that, unlike many production-based systems, individual productions in Soar do not correspond to its operators. Instead, the knowledge about when to apply an operator and how to apply it is spread out among a number of productions; thus the effect of an operator can vary considerably depending on the state to which it is applied.

Soar's processing is organized around a sequence of "decision cycles" (4). In each decision cycle, all the productions whose left-hand-sides match working memory are fired in parallel; this process is called "elaboration" (5). Since the changes that result may trigger additional productions, the decision cycle will proceed through a series of parallel production-firings, until "quiescence" is reached when no more productions fire. The productions actually carry out two distinct tasks: they implement the operator that was selected in the previous cycle, and they also make proposals about which operator to apply next. Thus the productions are simultaneously carrying out the previous decision and gathering proposals for the next decision. When quiescence is reached at the end of a decision cycle, Soar attempts to decide what to do next (6). If the current operator has been successfully applied,[4] and only one new operator has been proposed, the new operator is selected and Soar proceeds with the next decision cycle.

If the current operator could not be applied, no new operator has been proposed for the next cycle, or more than one has been proposed with no way to decide between them, then Soar reaches an impasse (8) and creates a subgoal (9) to resolve the impasse. The new subgoal triggers productions that select and set up an appropriate sub-space for dealing with the impasse. Operators are then applied in the subspace until the condition that prevented Soar from proceeding in the superspace is resolved (e.g. in (s) Soar figures out which operator to apply next). Once the impasse is resolved, Soar removes the subgoal and continues processing in the higher problem space (3).

Thus the basic structure of Soar is to select and apply a sequence of operators in an attempt to reach some desired state. In the top space (3), the desired state(s) depend on the task Soar is working on. In sub-spaces (9), the desired state is one in which the impasse blocking Soar from proceeding in the superspace is eliminated. In addition to simply resolving impasses, subgoals allow Soar to learn new productions via "chunking" (11). Soar builds new productions (called "chunks") whenever processing in a subgoal produces a result in the state of a higher goal (i.e. adds or removes an attribute-value pair). The left-hand-side of the chunk contains any attributes or values in the higher state that were used in the subgoal processing that led to the result. Thus chunks record the work done in the subgoal, allowing Soar to produce the same result directly in subsequent cases, bypassing the impasse and subgoal that led to them. Chunking effectively moves the knowledge used for problem-solving in the subgoal up into a higher space; Soar will subsequently be able to use this knowledge automatically

[4]This must be explicitly indicated by a production that recognizes the conditions characterizing successful complete application of the operator.

(or "recognitionally") without having to do deliberate problem-solving in a sub-space.

3 NL-Soar: Models and Operators

Since NL-Soar is responsible for carrying on a conversation, it needs to maintain models of the ongoing discourse as well as the semantics and syntax of individual utterances it is comprehending and producing. Four models are produced:

Utterance Model (or u-model) This models the syntactic structure of an utterance, using structures from Government and Binding Theory [Cho81]; each utterance is represented by a tree whose links are labeled with syntactic relations such as head, specifier, and complement, and an explicit linear ordering on the leaves. Utterances that are not yet complete may be modeled by several GB tree structures and/or have some of the leaves of the tree filled by pointers to objects in the situation model.

Situation Model (or s-model) This models the objects, properties, and relations discussed in the discourse, i.e. a representation of the semantics of what is being said.

Discourse Model (or d-model) This models individual discourse turns.[5] For each turn, the model indicates the type of discourse move being made, the speaker and intended hearer, the content, and the corresponding structures in the situation-model and utterance-model.

Discourse Segment Model This models the overall discourse the agent is currently engaged in. The discourse segment model keeps track of the participants in the discourse and a history of the discourse moves they have made,[6] the situation-model objects that have been introduced into the discourse, and the goals the agent is trying to achieve through the discourse.

NL-Soar manages the generation process by successively applying operators that make small modifications to the various models, gradually working towards the decision of what word(s) to actually produce. The operators used by generation currently include:

Discourse move operators These operators implement decisions about what sort of discourse move the agent wishes to make next. They generate a new discourse move in the discourse segment model and make any other appropriate modifications to that model. For example, the accept-discourse-segment operator can be used to indicate willingness to participate in a conversation someone else has started; whereas the open-discourse-segment operator initiates a new conversation.

d-generate This operator constructs a new discourse turn in the d-model to implement a move in the discourse-segment model.

d-realize This operator realizes a d-model element as one or more s-model and/or u-model elements, i.e. as a set of semantic and/or syntactic structures.

s-realize This operator realizes an s-model object as a (possibly partial) u-model structure.

say This operator releases a word to the motor system to be printed and/or spoken.

The following section will demonstrate how these models and operators are used to generate an utterance.

4 Managing a Conversation: NTD-Soar and NL-Soar

The first system to make use of NL-Soar's generation component is NTD-Soar. NTD-Soar is a system designed to simulate the activities of the NASA Test Director, the person responsible for co-ordinating the activities involved in the launch of a space shuttle [NLJ94]. The NTD's task involves (among other things) following along the planned launch steps in a manual, watching a set of television monitors that display pictures of the launch pad and related facilities, and communicating with other members of the launch team over a multi-band radio. Thus NTD-Soar must integrate its use of NL-Soar with the other activities it is engaged in.

The examples here come from NTD-Soar's modeling of the following conversation, taken from a transcript of an actual launch attempt (CVFS and FLT are two other members of the launch team):

```
Speaker   Utterance
CVFS      NTD, CVFS
NTD       Go ahead, CVFS
CVFS      Ready for BFS uplink
NTD       I copy
NTD       Houston Flight, NTD
NTD       Perform BFS preflight uplink loading
FLT       In work
```

The processing by which NL-Soar produces "Go ahead, CVFS" begins with the activity shown in the following trace. The trace shows (in a form simplified for readability) the sequence of decisions made at the end of each decision cycle; in general, this is an operator that Soar has decided to apply during the next cycle. NL-Soar is running recognitionally here, i.e. with all necessary knowledge built into chunks that fire in the top space. We will see in subsequent examples what

[5]Actually, only the current and previous turn are kept in working memory at this level of detail.

[6]The representation of the discourse moves at this level is more abstract than in the discourse model, containing only the type of the move and the speaker.

happens when NL-Soar reaches an impasse and must drop into a subspace.

```
...     (NTD-Soar performs various tasks
...      and comprehends "NTD, CVFS")
57:     accept-d-segment opened by "NTD, CVFS"
58:     d-generate discourse move 'answer'
59:     d-realize 'answer'
```

At the beginning of decision cycle 57, NL-Soar's comprehension code has built the discourse segment model shown in Figure 2 to represent the summons from the CVFS.[7] Here the discourse segment model represents the summons that opened the discourse, with pointers into the situation model to indicate the participants.

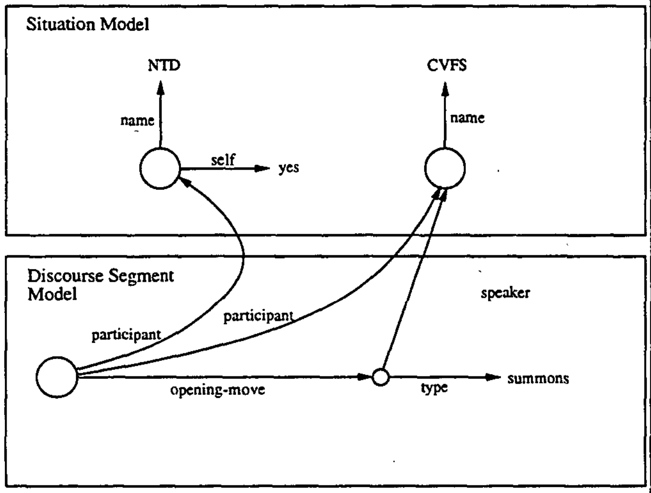

Figure 2: The models after comprehending "NTD, CVFS"

The existence of the discourse segment model prompts NL-Soar to consider discourse move operators for various ways to respond; since the NTD is required to explicitly acknowledge any communications, the accept-discourse-segment operator is chosen. Application of this operator modifies the discourse segment model as shown in the bottom portion of Figure 3. A new turn has been added to represent the NTD's response, and the discourse segment has been marked as accepted to indicate that the NTD has decided to participate in it.

Next, the d-generate operator selected in decision cycle 58 builds a d-model representation of the NTD's utterance, followed by a d-realize operator that builds a u-model and s-model realization of it; NL-Soar bypasses the s-model to directly build a u-model here because the phrase "go ahead" directly expresses the discourse move "answer" (in the specialized sub-language of this domain). The result of the operators selected in cycles 57–59 is the set of models for the utterance under construction shown in Figure 3. The u-model structure represents the partially-formed utterance "<NTD>, go ahead <CVFS>", where <NTD> and <CVFS>, are s-model objects that need to be expanded into u-model structures by subsequent operators.

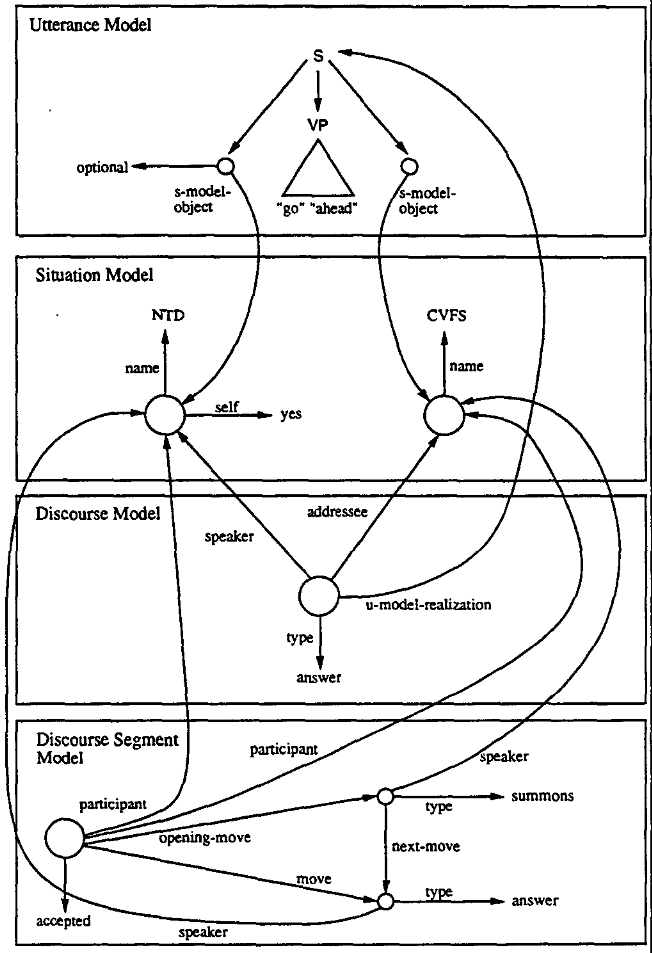

Figure 3: The models after applying the d-realize operator

NL-Soar then proceeds to finish construction and output of the utterance:

```
60:     s-realize <NTD> as adjoin of S
61:     say "go"
62:     say "ahead"
63:     s-realize <CVFS> as adjoin of S
64:     say "cvfs"
...     (NTD-Soar proceeds with other tasks)
```

The s-realize operator selected in decision cycle 60 "realizes" <NTD> as a null phrase, because it is marked as optional.[8]

[7]The utterance model and discourse model built for "NTD, CVFS" are not shown here because the generation operators don't make use of them.

[8]This decision, as with many others in this example, is made via chunks built up previously by processing in various subspaces. If these chunks had not

The next two items in the utterance are words, so NL-Soar simply uses say operators to output them. Finally, the s-realize operator selected in decision cycle 63 produces "CVFS", which is immediately output by the subsequent say operator. The utterance is now complete.

This example is a fairly simple one, but it demonstrates the basic top-level structure that NL-Soar uses for generation. The example doesn't show any interleaving of generation with other tasks, because the current implementation of NTD-Soar gives generation operators priority over task operators. Note, though, that the division of the generation processing into discrete operators that perform individual steps allows task operators to be interleaved with the generation operators unproblematically (as will be shown below). In addition, the processing here is entirely driven by NL-Soar's language code; there is no task goal driving the generation (although it was task knowledge that made the decision in cycle 57 to accept the summons rather than, say, ignoring or rejecting it.) This too is not generally the case, as the next example will show.

5 Interactions between task and NL

The following example shows a later portion of the conversation in the previous section; here the NTD has closed off his conversation with the CVFS and is now proceeding to talk to Houston Flight:

```
...         (NTD says "I copy")
153:     new-task contact-flt-for-bfs-uplink
154:     open-discourse-segment
```

In the decision cycles leading up to 153, the NTD has been informed that the CVFS is ready for the BFS uplink. In cycle 153, NTD's task operations recognize that the NTD must now tell FLT to start the BFS uplink. The new-task operator carries this out by posting a communicative goal to order FLT to do so; it is the presence of this goal that triggers NL-Soar's generation component.

Since the NTD is not currently talking to FLT, NL-Soar applies an open-discourse-segment operator to open up a conversation. Decision cycles 155 through 160 then proceed in a manner similar to the previous example, producing the utterance "Houston flight, NTD":

```
155:     d-generate discourse move 'summons'
156:     d-realize 'summons'
157:     s-realize <FLT>
158:     say "houston flight"
159:     s-realize <NTD>
160:     say "ntd"
```

Before proceeding to generate its next utterance, NL-Soar must wait until FLT explicitly acknowledges the NTD's summons (this is a task-specific requirement). So for the decision cycles after 160, NTD-Soar selects wait operators, which don't do anything.

```
161:     wait
162:     wait
...
182:     wait
```

The important point here is that it is NTD-Soar's *task* knowledge that is selecting the wait operators, not any part of NL-Soar. In decision cycles 161-182, NL-Soar doesn't propose any operators (because it's waiting for FLT's acknowledgement), so task operators are adopted even though NL-Soar is still in the middle of responding to a communicative goal. As it happens, the NTD doesn't have any other current tasks to carry out, so it simply selects wait operators. If there were other tasks the NTD needed to perform, though, they could be carried out during these cycles without interfering with NL-Soar's work.[9]

Finally, in decision cycle 183, the NTD becomes impatient and proceeds to simply assume that FLT has heard the summons and is ready to continue.[10] This is modeled by adding an <impatient> property to NTD-Soar's representation of the NTD if it has been waiting longer than a specified time; this in turn triggers the assumption that the FLT has implicitly acknowledged the summons, indicated by adding an <accepted> flag to the discourse segment model (as was done when the NTD responded to the CVFS's summons in the example above). This allows NL-Soar to resume working, and in decision cycles 183-189 it proceeds to generate "Perform BFS preflight uplink loading", thus achieving the original goal posted by the task back in decision cycle 154:

```
183:     continue-discourse-segment
184:     d-generate discourse move 'directive'
185:     d-realize 'directive'
186:     s-realize <do-uplink> as top-level
187:     say "perform"
188:     s-realize <uplink>  as comp of V
189:     say "BFS preflight uplink loading"
```

This conversation demonstrates how NL-Soar allows language processing to be integrated and interleaved with other tasks. First, the NTD acquires information from its conversation with the CVFS, namely that a particular step in the launch preparation is ready to be carried out. The NTD then decides that its next task should be to go ahead with the launch step, so it should tell FLT to carry it out. Doing this requires communicating in language, so NL-Soar starts applying operators to generate the necessary utterances. When NL-Soar doesn't propose any language operators (because it's waiting for a response), NTD-Soar can invoke other task operators to handle whatever else it needs to do at the time. Thus language

yet been built up, the s-realize operator here would reach an impasse, driving NL-Soar into a sub-space in which it would perform deliberate reasoning about how to carry out the realization, as in the examples in Section 6.

[9] A previous version of NTD-Soar did, in fact, return to visually scanning the manual page during this interval.

[10] This follows the real behavior in the transcript, in which the NTD goes on after a pause even though he's supposed to wait for the acknowledgement; this is actually the most common deviation from NASA's official language protocols that occurs in the transcripts.

and other tasks interact and can be interleaved as the situation allows.[11] This integration is still on a fairly large scale, though; in the next example we will see how language and task operations can be interleaved on an operator-by-operator basis.

6 Interleaving and interruption: TacAir-Soar and NL-Soar

TacAir-Soar is a system that simulates a fighter pilot [RJJ+94]. TacAir-Soar flies a (simulated) plane, controls its radar and weapons, and also communicates with other planes.[12] TacAir-Soar is being integrated with NL-Soar to handle its communication tasks. Because TacAir-Soar is engaged in a task where rapid response can be critical, it requires a more fine-grained interleaving of language with its other operations than NTD-Soar. In particular, the default assumption that language operators proposed by NL-Soar should have the highest preference, made in NTD-Soar, is not tenable here. Instead, TacAir-Soar's (current) default is to choose between language and other operators randomly; thus language and flying the plane can both proceed as a default, while still allowing specific situations to override the default and force TacAir-Soar to concentrate on critical maneuvers.

The following example shows how TacAir-Soar can interleave language and task operators. Here "Parrot101" is the call-sign of the plane TacAir-Soar is flying, and "Ostrich" is the name of another plane supporting Parrot101. Unlike NTD-Soar, TacAir-Soar maintains a stack of subspaces for long periods of time, because it uses operators that perform lengthy operations (e.g. the top-space operator for most of a mission is simply execute-mission). As a result, NL-Soar's language operators must be able to fire in any of TacAir-Soar's spaces, because the top-space already has an operator that can't be replaced. The traces for TacAir-Soar are therefore slightly more complex than the ones for NTD-Soar:

```
...     (TacAir-Soar flying to waypoint)
...     (TacAir-Soar detects unknown plane)
19      ask-for-bogey-id
20      look-for-commit-criteria
21      open-discourse-segment
```

In the processing before decision cycle 19, TacAir-Soar has detected a "bogey" (i.e. an unknown plane) on its radar. In cycle 19, the ask-for-bogey-id operator is selected to ask Ostrich to identify the bogey; this operator posts a communicative goal to request the information. Meanwhile, though, TacAir-Soar has other things to do, so in decision cycle 20 it decides to check if the criteria for committing to responding to the bogey have been satisfied.[13] Then, in cycle 21, it decides to use an open-discourse-segment operator to start talking to Ostrich. The next few cycles are similar to the previous examples:

```
21      open-discourse-segment
22      d-generate discourse move 'summons'
23      d-realize 'summons'
24      say "ostrich"
25      say "this"
26      say "is"
```

The operators in decision cycles 22–26 produce most of the summons "Ostrich, this is <plane-call-sign>". But then starting in cycle 27, more task operators fire even though NL-Soar hasn't completed the utterance:[14] (The indentation here indicates Soar when drops into a subspace because of an impasse.)

```
    26        say "is"
    27        check-commit-criteria
Commit criteria satisfied. Stopping Fly-To-Waypoint.
    28    intercept
    29    ==>
    ...
    32        select-intercept-geometry
    33:       ==>
    ...
    36            choose-intercept-altitude
        OUTPUT: Set desired-fpa to 10
```

In decision cycle 27, TacAir-Soar detects that its commit criteria have indeed been satisfied, so it then decides in decision cycle 28 to apply an intercept operator in order to intercept the bogey. TacAir-Soar can't do this in the current problem-space, so it drops down into a subspace in which it decides (in cycle 32) to apply a select-intercept-geometry operator. This in turn impasses into another sub-space in which TacAir-Soar applies a choose-intercept-altitude operator which sets the desired altitude to 10,000 feet. At this point, time is available for language to continue, so language operators fire in the current task space:

```
37          s-realize <Parrot101> as
                      adjoin of S
38          ==>
...
42              realize-by-name
43              name-is-known
44              category-match: NP adjoin S
45              return-result
```

In decision cycle 37, while in the middle of trying to intercept the bogey, TacAir-Soar decides to continue with its current utterance, attempting to express <Parrot101>. In this case, though, the necessary knowledge to implement the s-realize operator has not yet been compiled into a chunk, so Soar drops down into a language sub-space. The sub-space implements the s-realize operator by selecting a realization strategy (e.g. realize-by-name, realize-by-pronoun, or realize-lexically). If

[11] This also allows the task to cut off generation or alter what's being said in the middle of an utterance by changing the goal structures driving NL-Soar, although we have not yet implemented this.

[12] These can be other copies of TacAir-Soar or planes controlled by other simulation programs. The planes are connected via ModSAF [CSC+93], a separate program which provides each plane with information about the simulated world they are all flying in.

[13] Note that these don't include having identified the bogey.

[14] Some details of the processing that aren't relevant to the discussion are omitted here and in subsequent portions of the trace.

the selected strategy produces a syntactic structure that satisfies all relevant constraints (which can be syntactic, semantic, or pragmatic), it is returned to the top-space. If not, other strategies are attempted until one succeeds.

Here the realize-by-name strategy, chosen in cycle 42, builds the noun-phrase "Parrot101", which passes the relevant constraints. In cycle 45, Soar returns this result, thus resolving the impasse that arose in cycle 38.

Before a say operator can be selected for "Parrot101", though, TacAir-Soar invokes some further task operators:

```
...
49      search-last-position
    OUTPUT: Set desired-heading to -93
50      ==>
...
53          say "parrot101"
```

Task operators continue to fire through decision cycle 52. Finally, in decision cycle 53, the say operator is selected and the utterance is completed.

What this example demonstrates is how task and language operators can be interleaved in a very flexible way. NL-Soar was initially triggered by a communicative goal posted by the ask-for-bogey-id operator selected in decision cycle 19; an additional task operator, however, was applied before NL-Soar began its work with the open-discourse-segment operator in decision cycle 21. After a series of language operators fired, a task operator was selected in decision cycle 27 even though the utterance was not yet complete. Additional task operators were then selected until decision cycle 37, when the s-realize operator was finally selected. When this operator had completed (which required work in a subgoal to resolve an impasse), still more task operators were selected in decision cycles 46–52; finally the say operator selected in decision cycle 53 finished the utterance.

In addition to this fine-grained interleaving, TacAir-Soar can in fact interrupt language operators that are not yet complete to select task operators when necessary, as in the following trace:

```
72      s-realize <detect-on-radar>
73      ==>
...
77          realize-lexically
78          ==>
79              Problem-space: lexical-choice
...
81              pick-head
82              ==>
...             [generation continues working]
110 lock-radar-for-missile
... [further task operations]
```

Here NL-Soar attempts to apply an s-realize operator to express <detect-on-radar>.[15] This leads to an impasse in decision cycle 73, with further processing going down into several sub-spaces as NL-Soar attempts to carry out lexical choice. Then, in decision cycle 110, TacAir-Soar makes a decision in a problem-space higher-up in the goal stack to apply another task operator. Specifically, it decides to lock its radar on the enemy plane in preparation for shooting at it. Selecting this operator causes Soar to lose all pending work at a lower level of the goal-stack, which in this case includes the s-realize selected in cycle 72. In essence, TacAir-Soar has interrupted NL-Soar's processing in order to proceed with shooting a missile. This is just the kind of interruptability a real-time system needs to have; if something needs to be done right away (e.g. shooting a missile), it must be able to interrupt less critical activities in progress (e.g. language generation). When critical task operations have been completed, NL-Soar can re-select the s-realize operator and carry on with generation. Note that not all of NL-Soar's work has been lost; the language operators before cycle 72 had already completed, so the models reflect their changes. Furthermore, any chunks built in the sub-spaces before the interruption will fire when the s-realize operator is reselected, so some of the processing in cycles 73–110 won't need to be repeated. At worst, NL-Soar will only have to repeat a single (top-level) operator; but the fine-grained interleaving NL-Soar is designed around ensures that this will be a relatively small amount of work.

7 Reactivity, Learning, and Operator Size

The discussion so far has glossed over an important question: how much work should each operator do? Deciding that NL-Soar should generate incrementally doesn't necessarily imply what the size of the incremental steps should be. There are some obvious general guidelines, of course. Each operator must be "large" enough to do *something*. Conversely, operators must not be so large as to overcommit Soar to a particular utterance; a single operator that generated a ten-minute speech would be too large because it would leave Soar no way to make even a small modification to the speech to reflect its current situation.[16] These guidelines leave a lot of freedom to choose operator sizes; it would be useful to have some further criteria to constrain the operators.

Soar in fact provides further constraints on operator size through its learning mechanism. Whenever Soar is unable to completely apply an operator,[17] it creates a subgoal to figure out how to complete the operator. The results of the processing in the subgoal are compiled into productions called "chunks"

[15] This will eventually lead to "I have a contact."

[16] Soar could still interrupt the utterance at some point and discard the unspoken remainder. But modifying the speech in any way would mean redoing much of the work that originally built it, because the intermediate steps in its construction would have been compiled out.

[17] Recall from Section 2 that completion of an operator must be explicitly asserted by a production.

that are added to Soar's long-term memory. The conditions in the left-hand-side of the chunks generalize from the particular structures currently in working memory. Thus the chunks will apply in other similar situations, allowing Soar to transfer what it has learned from the impasse in the current situation.

The possibility of transfer of knowledge depends, though, on the size of the operator. A very large operator that integrates a lot of problem-solving in a subgoal is likely to lead to chunks that are very specific, because they depend on and affect a lot of working memory; they will therefore not transfer to other situations very often. On the other hand, very small operators that do only a little work will not gain much from chunking, because the chunks will compile only a small amount knowledge, preserving the need to apply a large number of operators at the top level. There is a tradeoff between large operators that are very likely to have impasses (because few previous chunks will transfer to them) and small operators that won't allow much learning. The use of chunking thus pushes NL-Soar towards "medium-size" operators that have a moderate amount of transfer and whose chunks collapse a moderate amount of processing.

There is a similar tradeoff between chunking and reactivity. The more work a single operator does, the faster NL-Soar will be able to generate utterances (assuming the operator doesn't reach an impasse). On the other hand, an operator that does a lot of work will need a lot of processing in a subgoal (or subgoals) to build its implementation, and the more time NL-Soar spends in the subgoal that builds the operator, the greater the chance that some other task will interrupt NL-Soar and replace the top-level operator without resolving the impasse and building the corresponding chunks. At the least, this will mean repeating work; in the worst case there might operators that never get enough time to finish. The tradeoff here is really between before and after chunking has occurred: larger operators will run faster once the necessary chunks are built, but are more likely to take too much time to get built in the first place.

These tradeoffs provide a more refined answer to the question of operator size. Operators should be designed so that their pre-chunking implementation takes a moderate amount of time and will produce general but non-trivial results. This will provide the reactivity needed by a real-time system in both pre-chunking and post-chunking situations; indeed, the specific real-time requirements of a particular application can determine the appropriate operator size.

8 Conclusion

The primary challenge for NL-Soar's generation processing is to perform generation in a way that allows flexible shifting of control between language and task operations. As we have seen, NL-Soar provides this flexibility by organizing generation as a sequence of incremental steps that can be interleaved with task actions as the situation requires. The result is a language generation capability that can be integrated with task operations at a broad level (as we did in NTD-Soar) or a more fine-grained level (as we did in TacAir-Soar); in critical situations, task operations can even interrupt language operators. This interleaving is driven by the requirements of the situation in which the system is operating; when there are no time-critical non-linguistic tasks pending, language can proceed uninterrupted. Thus NL-Soar provides the flexible control a system operating in a real-time environment needs.

Acknowledgements

We wish to thank Greg Nelson and Rick Lewis for extensive help and feedback during the integration of NL-Soar with NTD-Soar.

References

[Cho81] Noam Chomsky. *Lectures on Government and Binding*. Foris Publications, Cinnaminson, NJ, 1981.

[CSC⁺93] R. B. Calder, J. E. Smith, A. J. Courtemanche, J. M. F. Mar, and A. Z. Ceranowicz. ModSAF behavior simulation and control. In *3rd Conference on Computer Generated Forces and Behavioral Representation*, 1993.

[Lew93] Rick Lewis. *An Architecturally-based Theory of Human Sentence Comprehension*. PhD thesis, Carnegie Mellon University, 1993. Also available as Technical Report CMU-CS-93-226.

[LLN91] Jill Fain Lehman, Rick Lewis, and Allen Newell. Integrating knowledge sources in language comprehension. In *Proceedings of the Thirteenth Annual Conferences of the Cognitive Science Society*, 1991.

[LNR87] John E. Laird, Allen Newell, and Paul S. Rosenbloom. Soar: An architecture for general intelligence. *Artificial Intelligence*, 33:1–64, 1987.

[New90] Allen Newell. *Unified Theories of Cognition*. Harvard University Press, Cambridge, MA, 1990.

[NLJ94] Greg Nelson, Jill F. Lehman, and Bonnie E. John. Experiences in interruptible language processing. In *Proceedings of the 1994 AAAI Spring Symposium on Active NLP*, 1994.

[RJJ⁺94] Paul Rosenbloom, Lewis Johnson, Randy Jones, Frank Koss, John Laird, Jill Lehman, Robert Rubinoff, Karl Schwamb, and Milind Tambe. Intelligent automated agents for tactical air simulation: A progress report. In *4th Conference on Computer Generated Forces and Behavioral Representation*, May 1994.

[RLN93] Paul Rosenbloom, John Laird, and Allen Newell. *The Soar Papers: Research on Integrated Intelligence*. MIT Press, Cambridge, Massachusetts, 1993.