4.1: TacAir-Soar Code Layout, at the Problem-space Level

Paul E. Nielsen

May 2, 1996, 8:45-9:30am

Soar/IFOR Tutorial

University of Michigan AI Lab

Overview

This talk covers the layout of the TAS code according to the problem spaces.

First we will look at the directory structures.

Then we will look at some of the TAS problem spaces and their states.

The entire TAS problem space set is too large to list here, but you can use netscape to view it at http://ai.eecs.umich.edu/ifor/DDD/tacair/
Hierarchical Problem Spaces

The hierarchical problem spaces follow the operator goal hierarchy you’ve seen before.

Basic organization of problem space

- Code organization follows problem space hierarchy
- Documentation structure follows PS and code structure

For easier access, the top level directory has symbolic links to all such sub-directories.

Simple example:

- Top-goal → Execute-mission → Racetrack → Fly-racetrack-leg

Directory structure

By convention, when an operator has a sub-state

- A sub-directory will be created in the same directory as the operator
- The name of directory will be the name of the operator
- The name of the problem space will also be the name of the operator

The sub-directory contains

- File loading information for the sub-state
- All elaboration files for the sub-state
- All operator files for the implementing sub-state

This strict sub-state ⇔ sub-directory hierarchy covers about three fourth of the problem spaces in the TAS code. The rest are exceptions.
Exceptions to Hierarchical Problem Spaces - Floating Operators

Floating operators can trigger in any problem space as long as their preconditions are satisfied.

They are all in the **any-ps** directory.

For example:

- **authentication**: Provide coded authentication
- **change-division-lead**: Change the lead of a 4-ship
- **change-formation**: Change flight formation
- **communicate-winchester**: Out of weapons
- **give-existence-to-partner**: Tell partner your location
- **identify-contact**: Try to identify a bogey
- **message-repetition**: Repeat previous message

More Floating Operators - Interpret Situation

There are some special floating operators in **any-ps/interpret-situation**

These have to be floating operators because the world changes independently of what you are thinking.

Some examples of these are:

- **bogey-is-jinking**: Notice an evasive maneuver
- **in-bogey-lrm-range**: Within missile range
- **look-for-commit-criteria**: Okay to intercept
- **my-missile-is-stupid**: Missile missed
- **notice-bogey-fired-missile**: This is important
- **threat-is-gone**: All clear
More Exceptions - Shared Operators

There are a number of operators which can be proposed in more than one problem space.

These are found in **common**

They are not floating because they are still restricted to a limited number of problem spaces.

For example:

- **achieve-proximity**: Fly toward a specific agent
- **adjust-radar-elevation**: Slew radar as appropriate
- **join**: Join up into a flight formation
- **launch-missile**: Fire a weapon
- **strip**: Split up a flight formation
- **wait**: We do a lot of this
- **waypoint-computer**: Enter coordinates of a waypoint

More Exceptions - Parallel Hierarchy

The correct way to manage multiple lines of reasoning in Soar is still a research issue.

In Soar/IFOR some of the controllers need to do this because they are executing a separate flight mission and control mission.

Our solution is to implement the control mission as a collection of floating operators under **execute-mission/cas-control**.

Rather than reaching an impasse, the control operators will mark the incomplete information on the top state and terminate.

Other operators will be invoked to flesh out the mission.
Parallel Hierarchy Example

- **inbound-control**: TAD or DASC clears inbound flights
- **assemble-orders**: Determine information to provide
  - **mission-changes**: Inform there are mission changes
  - **read-brief-card**: Give changes in brief form
  - **read-mission-brief-line-N**: Read individual lines

Load Control Variables

The Soar/IFOR code is generic.

All Soar/IFOR agents could potentially load all of the code with no noticeable affect on their behaviors.*

For faster initialization we tag portions of the code and load only those portions necessary.

Specific capabilities enabled by loading specific portions of code

For example:

- **radar**
- **weapons**
- **flying**

*In fact, the version used for this tutorial does load all the code for all agents.
Inside the Problem Spaces

That should give you a feel for the problem space code layout.

Now we are going to look inside a few of the problem spaces in some more detail.

The Top Problem Space

The top problem space can be thought of as the global level of a program.

Some of the things which happen in top-ps include:

- **init-agent**: Prepare data structures for the agent
- **init-plane** or **init-vehicle**: Prepare the vehicle’s data structures
- **select-mission**: Select the current “big” mission
- **plan-mission**: Plan out the mission
- **execute-mission**: Carry out the selected mission
- **terminate-agent**: Quit when hit or mission completed
Top-level state map

top-ps maintains a number of data structures which can be accessed from any other state in a Soar program.

This includes:

- The input and output links

- Default symbolic values including missile, radar, plane, section information
  - e.g., \u2192 \texttt{radar.elevation.value}

- Elaborations describing bogeys
  - e.g., \u2192 \texttt{target.group.member}
    \texttt{agent.roe.achieved}

- Communication elaborations
  - e.g., \u2192 \texttt{comm.message.content.item}

Top-level state map - continued

- Elaboration for controlling the planes
  - \u2192 \texttt{control.group.member.radio}

- Elaborations for a route flying
  - \u2192 \texttt{flight-plan.leg.id.start}

- Elaborations for flying in formation
  - \u2192 \texttt{formation.lateral-separation.value}

- Mission elaborations
  - \u2192 \texttt{mission.altitude.value}

- Miscellaneous elaborations
  - There is a map of the top state at
    \url{http://ai.eecs.umich.edu/ifor/DDD/tacair/top-ps/state.map.text}
The Execute Mission Problem Space

Once an agent starts executing it’s mission, it will maintain that problem space until it successfully returns to the home base or the agent is destroyed.

The sub-goals of execute mission implement various parts of the overall mission.

Some of the things which could happen in `execute-mission` include:

- `change-mission`: Modify current mission
- `fly-flight-plan`: Fly a route
- `follow-leader`: Stay in formation
- `intercept`: Intercept a bogey or bandit
- `racetrack`: Fly circular pattern
- `return-to-base`: Go home
- `strip`: Split up

The Fly Control Point Problem Space

`fly-control-point` is an example of a more typical problem space in Soar/IFOR.

The focus is on performing a more specific task than executing an entire mission.

During the course of flying a route, here are all the things a plane could do when it reaches a control point:

- `accept-permission`: Receive permission to leave
- `circle`: Stay put
- `communicate-contact`: Contact control agent
- `communicate-inbound`: Say we’re inbound to target
- `communicate-strike`: Tell strike package to leave
- `communicate-push`: Tell controller we’re leaving
- `fly-delay`: Kill some time
- `refuel`: Take on fuel
- `waypoint-computer`: Set computer for next waypoint
The Fly Control Point Problem Space

The state elaborations for flying a control point (in addition to io-state, name and problem-space) are:

- ^point
  - Purpose: provide immediate access to the current point
  - Created by: state initialization
  - Tested by: All operators

- ^communicated-point *yes*
  - Purpose: note that point was communicated to contact
  - Created by: communicate-contact
  - Tested by: terminate-goal

The Fly Control Point Problem Space - continued

- ^communicated *yes*
  - Purpose: remember that communicated with wingman
  - Created by: communicate-point
  - Tested by: terminate-goal

- ^times-circled
  - Purpose: provide count of number of times circled
  - Created by: state initialization
  - Tested by: circle

- ^time-til-leave
  - Purpose: provide times for employing circles of various sizes
    - Created by: state initialization
    - Tested by: circle
Summary

There is a logical layout to the Soar/IFOR code which follows the problem space hierarchy.

We have seen some of the Soar/IFOR problem spaces.

We have seen some of the state definitions for the Soar/IFOR problem spaces.
Exercise 4.1: Working with Problem Spaces

Paul E. Nielsen

May 2, 1996, 9:30-10:15am

Soar/IFOR Tutorial

University of Michigan AI Lab

Exercise Goals

The goal of this exercise is to familiarize yourself with the existing Soar/IFOR problem spaces.

First we will look at the code to see how state information is defined.

Then we will run a scenario and see how it is created.

Later today you will learn to create new problem spaces.
The Problem Space Hierarchy

Start up **netscape** and go to the page which defines the Soar/IFOR goal/operator hierarchy:

http://ai.eecs.umich.edu/ifor/DDD/tacair/

Scroll through the goal/operator hierarchy.

What are the sub-goals of "racetrack"?

What is the super-goal of "racetrack"?

*You will need a password to read this.

---

The Racetrack Problem Space

Click on "racetrack" and read it's definition. Then follow some of the links to related information. You can

- Look at the super-goal
- Read the source material on the missions which fly a racetrack
- Explore the subgoals
- Investigate the actual Soar code which implements this operator
Floating and Common Operators

Go back to the Soar/IFOR goal/operator hierarchy

http://ai.eecs.umich.edu/ifor/DDD/tacair/

Find where floating and common operators are defined.

What are three “floating” operators? What do they do?
What are three “common” operators? What do they do?

The Top State Map

Go to

http://ai.eecs.umich.edu/ifor/DDD/tacair/
top-ps/state.map.text

You can get there by clicking “top-goal”, “state elaborations”, “state.map.text” from the goal/operator hierarchy page.

This page isn’t pretty, but it gives you a “complete” picture of all the information which can exist on the Soar/IFOR top state.

http://ai.eecs.umich.edu/ifor/DDD/tacair/
top-ps/top-ps-elab.html has explanations for some of these top state elaborations.

Look at “Elaborations describing bogeys” and “LS and TA elaborations”.

Local States

Go back to the Soar/IFOR goal/operator hierarchy
http://ai.eecs.umich.edu/ifor/DDD/tacair/

Click on “Fly-Control-Route”.

What information is on the local state?

Since route flying is complex, it will create a sub-state.

What information gets put on this sub-state?

Identify Problem Spaces

Run micro-TAS with the watch level set to 1 (default)

The names scrolling by represent operators and states

- What activity do you observe?
- What happens after an agent begins to execute it’s mission?

Stop Soar and print the goal stack

- What is the top state?
- What is the current state?
- What is the current operator?
- What are the super states?
Inspecting the State

With Soar stopped, use the "print" command to look at the current state, and the top state.

What is the current mission?

To follow links, you can either print successive working memory elements, or use the "-depth N" option to print.

Print the current mission to a depth of 3.

- What is the mission type?
- What is the mission number?
- What is the mission altitude?

Breaking Tasks into Problem Spaces

Where in the problem space hierarchy would the following go?

- Changing the mission?
- Targeting an enemy aircraft?
- Firing a missile?
- Maintaining formation?
4.2: Extending the Soar/IFOR Agent, Part One
Adding Productions to Existing Operators

Paul E. Nielsen

May 2, 1996, 10:30-11:30am

Soar/IFOR Tutorial
University of Michigan AI Lab

Overview

Productions are the fundamental statements of the Soar language.

This talk focuses on some of the details of programming Soar/IFOR agents, so it will be interleaved with exercises which detail how to add code to a micro version of the Soar/IFOR program.

At the end of this talk you will know:
- How to write IFOR productions independent of existing operators
- How to write productions which modify existing IFOR operator behaviors

This talk and the following exercise are combined since I believe the best way to learn programming is by writing code.
Review: Writing a Soar Production

The following syntactic form plus a few parens, up-arrows, minus signs, and periods is just about all the syntax you'll have to know for this talk/exercise.

```
sp {production-name
    CONDITIONS
    -->
    ACTIONS
}
```

Exercise Part 1: Introduction

The goal of the exercises in this session will be to implement a "Beam-CAP".

When the rules of engagement are very tight, fighters may shift from a Bar-CAP to Beam-CAP

- A Beam-CAP is just a racetrack oriented perpendicular to the oncoming bogey
- It provides some defense while allowing bogey to close

Micro TacAir Soar already has a Bar-CAP mission.

Which operator implements the Bar-CAP mission once the fighters are "on station"?
Naming Conventions for Productions

Soar/IFOR has adopted a few naming conventions to avoid naming conflicts and help locate a faulty production.

1. The first piece of the name should be
   - The problem-space name, if the production tests a problem-space name
   - The operator name, if the production tests an operator
   - Otherwise: default or some general context information

2. The second piece of the name should describe the function of the production
   - It may be an elaboration or a persistent-elaboration
   - It may propose, apply, or terminate an operator
   - It may also suggest-proposal conditions

More Naming Conventions

3. The rest of the name should reflect the intent of the production
   - The name of the augmentation being elaborated
   - The name of the operator being proposed
   - The particular case being handled during operator application
   - etc.

Examples:
- top-ps*elaborate*state*agent
- circle*propose*fly-second-third
- fly-second-third*apply

See:
http://ai.eecs.umich.edu/ifor/DDD/tacair/conventions/production-names.html
Exercise Part 2: Questions

What is a good name for our Beam-CAP production?

What information will you need for the LHS of your Beam-CAP production?

The LHS - Conditions

The condition side (LHS) of a production tests information on a state, either directly or indirectly, and if it matches the production will fire.

Finding the appropriate conditions for the left hand side of your function is the most difficult part of writing a production.

You can:

- Copy them from a similar production
- Look them up in the state map
- Get Soar to tell you
Review: State Organization

The local state is everything on the state attribute. This contains information required to solve current impasse.

- Local information
- Pointer to top state
- Pointer to super state

Every state has a top-state link that points to the top-state. The top-state contains information global to all problem spaces.

- Default symbolic values
- Descriptions of other agents
- Mission information

Information can be retrieved from or added to the super state from super-state.

---

Exercise Part 3: Exploring the State

Use the Exercise Editor to create a Bar-CAP mission and tighten up the commit criteria to something absurd.

Start up Micro-TacAir-Soar with your Bar-CAP mission.

Let it run until the fighters get on station, then stop it.

While Soar is stopped, take a few minutes to follow the links on the local state, top-state, and super-state. Try to find the following:

The current mission

- What is the current mission?
- What other information is available about this mission?
Exercise Part 3: Exploring the State Some More

Which weapons are available?

- How many are available?
- What other information is available about weapons?

Who is your wingman?

- Which groups does your wingman belong to?
- What other information is available about your wingman?

Can you see any other agents?

- Where are they?
- How do you know where they are?

Example - Trivial RECON

Here's a trivial production which prints "I see a plane" whenever it sees a new agent. It might be useful as a starting point for writing your production.

\[
\text{sp \{top-ps*new-contact} \\
\text{ `Notice new contacts`} \\
\text{ (state <s> ^problem-space.name top-ps} \\
\text{ ^agent <a>)} \\
\text{ (<a> ^contact <c>)} \\
\text{ (<c> ^active *yes*)} \\
\text{ -->} \\
\text{ (write (crlf) |I see a plane|)) } \}
\]

Explanation:
- Start with a state in the top problem space.
- Look for an agent which is an active contact.
- "write" is a function which sends it's arguments to the output stream.
Exercise Part 4: Bogey at 12 o’clock

Now use the ModSAF interface to put a bogey out near your fighters.

Let Soar run a few cycles to notice the bogey then stop it.

Look at the top-state again.

Is your agent aware of the bogey?

What info does your agent have on the bogey?

Is the information sufficient for the LHS of your Beam-CAP production?

Try to write the LHS of the Beam-CAP production.

Hint: Magnetic bearing is the angle between magnetic north and the direction to an object.

Exercise Part 5: Writing the First Production

Create a complete Soar production by adding a “write” statement, similar to one in the example, as the RHS.

Load your production into the agent’s window.

Turn up the watch so you can see what’s happening.

Run the your production. What happens?
The RHS - Actions

The action side (RHS) of a production may add or remove information from a state or link, or modify the io-link.

So, I lied. Finding the appropriate actions for the right hand side of your function can be just as difficult as finding the appropriate conditions for the left hand side. Especially if your production is trying to change something which already exists.

The great thing about Soar is that the solution is exactly the same for both problems. You can:

- Copy them from a similar production
- Look them up in the state map
- Get Soar to tell you

Creating a new link which doesn’t already exist is a much easier problem, so let’s try that first.

Exercise Part 6 : Adding a LHS

Write a production which adds a link called $^\wedge$beam-heading to the local state.

Use information about the bogey to compute the beam-heading direction.

Tip: There’s a RHS function, round-off-heading, which takes a heading and a step value. It returns a value rounded to the nearest step and converted to a heading (-179 to +180 degrees). Look it up.

Load and run your production. What happened?
Dealing with Ties

By default, Soar expects only one value for each link.

If you let your code run long enough, the bogey will move, and a second heading value will be computed.

When Soar tries to install the new value you get a tie.

If you didn’t get a tie impasse, you’re ahead of the game, but try the next exercise anyway.

Exercise Part 7: Testing for Existing Values

One way to resolve a tie is to include a test in the LHS which confirms that a value doesn’t already exist before trying to add it.

In this case we might note that we’ve already reacted to this agent and don’t need to take any further action.

For example:

- `^reacted-to <a> # The agent’s value is <a>`

The RHS should set this value. For example:

- `^reacted-to <a> # The agent’s value is <a>`
Exercise Part 7: More Testing for Existing Value

To avoid having to resolve the previous tie or reloading the code, rename `beam-heading` to `b-heading` in your production.

Now run your modified production. What happened?

If it was working without this condition, what’s happening now?

---

Review: Support

Soar has two ways to determine the persistence of a value

- I-support
  - Elaborations persist only as long as preconditions are valid
- O-support
  - Elaborations persist even if preconditions are no longer are valid

Generally, an elaboration gets O-support if the production tests an operator and a state, otherwise it gets I-support.*

---

*I’m over simplifying here. See the Soar manual for the full explanation.
Exercise Part 8: Revisiting the State

Is your production getting I-support or O-support?

How would you modify it to get the opposite form of support?

Next you have to track down the correct value to change which will have some effect on your CAP's orientation.

Hint: You don't need to change any values on the output link.

Preferences

So I lied again. There's one more bit of syntax you'll have to learn to write this production.

If a link already has a value and you want to change it there are two ways to do it:

1. You can use a reject preference. For example:  

   `<s>  
   ^link <old-value> -  
   </s>`

2. Or you can make the new value "better" than the old value or "best". For example:

   `<s>  
   ^link <new-value>  
   </s>`

What will happen if you give the new value a "best" preference and a another agent comes along?
Exercise Part 9: The Finished Production

Write a production which will change the orientation of an agent's CAP so that it's perpendicular to an incoming bogey.
4.4: Extending Agents, Part 2
New Operators

John E. Laird

May 2, 1996, 1:30pm-2:30pm

Soar/IFOR Tutorial
University of Michigan AI Lab

Review: Adding New Operators

- Operators are core of TacAir-Soar and the standard way to enhance the system.
- Productions are added for suggesting and existing operator, or modifying an implementation.
- Operators and subgoal (problem spaces) are used when a task is complex and can’t be performed as a single act (but then it is decomposed into individual operators).
Review: Four Parts of An Operator

1. Proposal: A production that tests the current state and creates an acceptable preference (+) for an operator. There may be many proposals for a given operator — different situations in which you want to do the same thing. We usually also have an indifferent preference (=). The preferences will have i-support.

2. Selection: A production that creates a preference (best, worse, ...) for an operator(s). We rarely use these in TacAir-Soar.

3. Application: Productions that make changes to the state based on the operator being selected. These changes will have o-support.

4. Termination: A production that creates a reconsider preference (@) for the selected operator. There may be many different terminations for a single operator.

An Example Operator: Attain-Intercept-Altitude

At the beginning of an intercept, we achieve an altitude in order to have an advantage later in the engagement. A simplified version of this operator is:

If we are already above the bogey, stay where we are. If we are below the bogey, climb to his altitude.

This is not always right — some pilots like to stay low against a MiG-29 because of its poor look-down radar capability.
Adding a New Operator: Total Picture

- Get to the right problem space directory: intercept.
- Create a new file: operator-name.soar
  (Attain-Intercept-Altitude.soar)
  Put in the productions.
- Add a source command to the load.soar file (source
  Attain-Intercept-Altitude.soar).
- Add a .html file to document the new operator
  (Attain-Intercept-Altitude.html).
- Add a pointer to that file in ../problem-space.html

These are done semi-automatically by using the templates in SDE.

Attain-Intercept-Altitude: Operator or Search Control?

1. Why should this be an operator rather than an production?
   - External actions should be operators (although sometimes violated as in adjust radar elevation).
   - We need to remember that we’ve done it.

2. Should it be a new operator, or just the selection of a climb operator?
   - We like to have the name of an operator be descriptive of how it is used so it is easier to tell what is going on from the trace.
   - We have common productions that can do the details of implementing all maneuvering operators.
Attain-Intercept-Altitude Preconditions

When should we propose the Attain-Intercept-Altitude operator?

- During an intercept.
- Threat is at an altitude higher than our altitude.
  Better might be higher than our desired altitude.
- We haven’t already done this for that altitude.

```
(state <s> ~problem-space.name intercept)

~threat-agent.state.position.altitude.value <alt>
~top-state.position.altitude.value < alt)

If we get fancy, it would be our desired altitude

~threat-agent.state.position.desired-altitude <alt>
~top-state.position.altitude.value < alt>

We haven’t already done this.

~intercept-altitude <alt>)
```
Attain-Intercept-Altitude Selection

When should we select the Attain-Intercept-Altitude operator?

- Whenever it is proposed (make it a best preference).
- Even if we are employing weapons? Probably not, so don’t terminate employ-weapons if this is proposed.
- Although we should probably terminate achieve-proximity if this is proposed. Need to add:

\[-\{(state <s> \text{^operator } <o> +)\) \\
\text{(}<o> \text{^name attain-intercept-altitude)}\}

\]

to achieve-proximity proposal.

Attain-Intercept-Altitude Operator

What should the Attain-Intercept-Altitude operator have on it?

- It’s name:

\text{^name attain-intercept-altitude}

- The desired-altitude:

\text{^desired-altitude }<\text{alt}>

- That it is an output operator and requires waiting:

\text{^type output }\text{^requires wait}
Attain-Intercept-Altitude Operator: code

This is if threat is higher than our current altitude.

\[
\begin{align*}
\text{sp } \{ & \text{intercept*propose*attain-intercept-altitude} \\
& \text{(state } <s> \text{ "problem-space.name intercept} \\
& \text{"threat-agent.state.position.altitude.value } <\text{alt}> \\
& \text{"top-state.position.altitude.value } <\text{alt}> \\
& \text{"-intercept-altitude } <\text{alt}> ) \\
\rightarrow & \text{(} <s> \text{ "operator } <o> \text{ + =, } > \\
& \text{(} <o> \text{ "name attain-intercept-altitude} \\
& \text{"desired-altitude } <\text{alt}> \\
& \text{"type output} \\
& \text{"requires wait} \\
& \} 
\end{align*}
\]

Attain-Intercept-Altitude Application

What should the Attain-Intercept-Altitude operator do?

- Because it is an output operator and has a desired-altitude means that some common productions will do the basic work.
- Need to record the intercept-altitude:
  \[
  (<s> \text{"intercept-altitude } <\text{alt}> )
  \]
- Need to remove any old intercept-altitudes:
  \[
  (<s> \text{"intercept-altitude } <\text{old-alt}> - )
  \]
Attain-Intercept-Altitude Application

\[
\text{sp \{attain-intercept-altitude*apply}
\begin{align*}
&\text{\quad (state <s> `operator <o>)} \\
&\text{\quad (\langle o \rangle `name attain-intercept-altitude} \\
&\text{\quad \quad `desired-altitude <alt>)} \\
\rightarrow \\
&\text{\quad (\langle s \rangle `intercept-altitude <alt>))} 
\end{align*}
\]

\[
\text{sp \{attain-intercept-altitude*remove}
\begin{align*}
&\text{\quad (state <s> `operator <o>} \\
&\text{\quad \quad `intercept-altitude <alt>)} \\
&\text{\quad (\langle o \rangle `name attain-intercept-altitude} \\
&\text{\quad \quad `desired-altitude <> <alt>)} \\
\rightarrow \\
&\text{\quad (\langle s \rangle `intercept-altitude <alt> -))}
\end{align*}
\]

Attain-Intercept-Altitude Termination

When is it done? As soon as achieved desired-altitude.

\[
\text{sp \{attain-intercept-altitude*terminate}
\begin{align*}
&\text{\quad (state <s> `operator <o>)} \\
&\text{\quad (\langle o \rangle `name attain-intercept-altitude} \\
&\text{\quad \quad `achieved desired-altitude)} \\
\rightarrow \\
&\text{\quad (\langle s \rangle `operator <o> 0}}
\end{align*}
\]
What went wrong?

- If enemy plane is climbing, continually trying to achieve its new altitude.
- Trying to achieve altitude keeps it from doing anything else (achieve-proximity and employ-weapons).
- To make more reactive, why not just terminate immediately (not wait to achieve)? But will still reselect each time enemy climbs!
- Why not just record that we did it once (not the exact altitude)? Will never adjust during a long intercept.
- What should we do?

Suggestions?

1. Terminate operator as soon as new altitude is specified (don’t wait for it to be complete).
2. Record not just the intercept altitude, but also a higher altitude to use as a boundary for reselection.

\[
\langle s \rangle \ ^\text{intercept-altitude} \langle ia \rangle \\
\langle ia \rangle \ ^\text{current-value} \langle alt \rangle \\
\ ^\text{high-value} (\ + \langle alt \rangle \ 300)) \\
\]

3. Propose if threat-agent’s position is greater-than that value.
Final Version: Proposals

sp {intercept*proposition*attain-intercept-altitude*first-time
 (state <> "problem-space.name intercept"
  "top-state.position.altitude.value < alt"
  "threat-agent.state.position.altitude.value < alt"
  "intercept-altitude")
 -->
  (<s> "operator <> + =, >)
  (<o> "name attain-intercept ALTITUDE"
     "desired-altitude <alt>
     "type output"
     "requires wait")}

sp {intercept*proposition*attain-intercept-altitude*second-time
 (state <> "problem-space.name intercept"
  "intercept-altitude.high-value < alt"
  "threat-agent.state.position.altitude.value < alt")
 -->
  (<s> "operator <> + =, >)
  (<o> "name attain-intercept-altitude"
     "desired-altitude <alt>
     "type output"
     "requires wait")}

Final Version: Application

sp {attain-intercept-altitude*apply
 (state <> "operator <>")
  (<o> "name attain-intercept-altitude"
     "desired-altitude <alt>")
 -->
  (<s> "intercept-altitude <ia>")
  (<ia> "current-value <alt>
       "high-value (+ <alt> 300))}

sp {attain-intercept-altitude*remove
 (state <> "operator <>"
  "intercept-altitude <ia>")
  (<ia> "current-value <alt>")
  (<o> "name attain-intercept-altitude"
     "desired-altitude <> <alt>")
 -->
  (<s> "intercept-altitude <ai> -)"
Final Version: Terminate

sp {attain-intercept-altitude*terminate
  (state <s> ~operator <o>)
  (<o> ~name attain-intercept-altitude)
  -->
  (<s> ~operator <o> 0)}

More interesting example: Beaming

- If an enemy plane might have shot a missile, a good counter measure is to go "beam" to his heading (90 degrees off) for 15 seconds.
- This can confuse the radar as well as put you in a good position to use chaff and flares (too bad ModSAF doesn't have them yet).
Beaming Proposal

When do we want to beam?

- There is a threat.
- If we are in range of a threat's missile envelope (LAR = launch acceptability region).
- If the threat is pointing at us.
- We aren't already beaming (or maybe we haven't beamed against this agent).

This should happen no matter what problem space we are in: this is a floating operator.

Beaming Proposal

- There is a threat.

  (state <s> "threat-agent <ta>")

- If we are in range of a threat's missile envelope (LAR = launch acceptability region).

  (<ta> "relative <rel>
  "state.weapon.missile.lar <lar>")
  (<rel> "slant-range.value < <m-range>
  "weapon-quarter <wq>")
  (<lar> "quarter <wq
  "slant-range-high <m-range>")

- If the threat is pointing at us.

  (<ta> "relative <rel>
  (<rel> "target-aspect.value < 9")

- We aren't already beaming (or maybe we haven't beamed against this agent).

  (<ta> "beamed *yes*"
Beaming Operator Proposal: Code

```
sp \{intercept*propose*beam
    (state <s> ^threat-agent <ta>)
    (<ta> ^^beamed *yes*
        ^relative <rel>
        ^state.weapon.missile.lar <lar>)
    (<rel> ^slant-range.value < <m-range>
        ^target-aspect.value < 9
        ^weapon-quarter <wq>)
    (<lar> ^quarter <wq>
        ^slant-range-high <m-range>)

-->
    (<s> ^operator <o> + =, >)
    (<o> ^name beam
        ^type output
        ^requires wait})
```

Beaming Operator Elaboration/Application

What do we need to add to the operator after it has been selected?

- The current time + 15 seconds so we know when to terminate.
- The desired-heading for the beam.

If these were added when operator was created, operator would be regenerated whenever they changed.
Beaming Operator Elaboration/Application

What do we need to add to the operator?

- The current time + 15 seconds so we know when to terminate.
- The desired-heading for the beam.

```lisp
sp {intercept*propose*beam
  (state <s> ^threat-agent.relative.magnetic-bearing.round <mb>
    ^top-state.current-time.value <ct>
    ^operator <o>)
  (<o> ^name beam
       ^beam-time)
  -->
  (<o> ^beam-time (+ <ct> 15)
    ^desired-heading (round-off-heading (+ <mb> 90) 1))
}
```

Beaming Operator Application

What does the operator do?

- Mark that the threat-agent is been beamed.

```lisp
sp {beam*apply*beamed
  (state <s> ^operator <o>
    ^threat-agent <ta>)
  (<o> ^name beam)
  -->
  (<ta> ^beamed *yes*))
```
Beaming Operator Termination

When is the operator done?

- When time is up.

```lisp
sp {beam*terminate
    (state <s> `top-state.current-time.value <ct>
    `operator <o>)
    (<o> `name beam
    `beam-time < <ct>)
    -->
    (<s> `operator <o> 0))
```

What’s Missing?

- Must interrupt Employ-Weapons!
- How?
- Have it terminate if beam is proposed.
- Modify proposal for employ-weapons. Add:

```lisp
-{(state <s> `operator <o1> +)
    (<o1> `name beam)}
```
4.4 and Exercise 4.4: Extending Agents, Part Three
New Problem Spaces

Randolph M. Jones

May 2, 1996, 3:45PM–5:30PM

Soar/IFOR Tutorial

University of Michigan AI Lab

Adding New Problem Spaces

Problem spaces can be used for many different purposes in Soar.
In the TacAir-Soar system, problem spaces are used exclusively to implement the ability to achieve tasks by dividing them into multiple subtasks.
When to Add a New Problem Space

If you want to add a new capability to an agent, and the capability is not simple enough to express as a single operator, consider adding an entirely new problem space.

Consider a problem space for tasks that:

- Consist of multiple actions
- Consist of multiple alternative actions
- Require sequential actions
- In sum, are difficult to implement in a single operator

Advantages of Using Problem Spaces

- Flexibility in implementing behaviors
- Explicit organization of knowledge
  - Can structure data in a manner appropriate to the goal
- Efficiency
  - Data structures can be maintained locally to the problem space
  - Switching between tasks within the same problem space is quick
Disadvantages of Using Problem Spaces

- Only one hierarchy of problem spaces can be active at a time
- Some actions are not neatly compartmentalized
- Efficiency
  - Invocation of a new problem space arises from an impasse and creation of a subgoal
  - These activities take time
- Must be careful that early interruption of the subgoal does not break anything

Details

Every problem space implements a super-operator

- Must specify proposal and termination conditions for super-operator
- Super-operator may or may not have application productions
- Must be careful about interactions between new super-operator and existing operators
- The problem space can make assumptions about what is true
- Generally want to specify a set of "elaborations" plus a set of operators for a problem space
- The overall "goal" of the problem-space operators is to terminate the super-operator
An Example Problem Space

Execute-tactic

The execute-tactic operator allows a section (flight of two) of aircraft to employ a particular, pre-briefed tactic during an intercept. An example of such a tactic is the "pincer".

This problem space is implemented in TacAir-Soar, but not in Micro-TacAir-Soar.

We will implement a new version in Micro-TacAir-Soar

Execute-Tactic Preconditions

When should we propose the Execute-Tactic operator?
Execute-Tactic Preconditions

When should we propose the Execute-Tactic operator?

- What problem space should execute-tactic belong to?
- Intercept
- Possibly somewhere else?
- We must have been pre-briefed with a tactic to execute
- Can only be initiated by the lead of the section
- It must be the appropriate time during the intercept to execute the tactic
  - Specified by tactic parameters
Execute-Tactic Code Location

Where will we put the actual files that hold the code for the Execute-Tactic operator and problem space?

- Put `execute-tactic.soar` in `intercept` directory
- Make `execute-tactic` subdirectory
- Problem space sub-operators go in `execute-tactic` subdirectory

Execute-Tactic Preconditions

```
sp {intercept*propose*execute-tactic
  (state <s> "problem-space.name intercept
   ^top-state.command.primary-group <cg>
   ^group <tg>)
  (<cg> "tactical.action <sta>)
  (<sta> "initiate-range <ir>
       "achieved *yes*)
  (<tg> "member.agent.relative.slant-range.round <= <ir>)
-->
  (<s> "operator <o> + >, =)
  (<o> "name execute-tactic
      "action <sta>
      "command-group <cg>
      "target-group <tg>)
}
```
Execute-Tactic Application

What should the Execute-Tactic operator do?

- Keep track of whether the tactic has been completed
- Subgoal activities
  - Carry out the tactic (more detail later)
- Terminate when the tactic is completed

Other details:

- Also must change proposal of the Employ-Weapons operator
Execute-Tactic Application

\[
\begin{align*}
sp \{ & \text{execute-tactic*apply*achieved} \\
  & (\text{state }<s> \text{ top-state.command.group.tactical.action }<\text{sta}> \\
  & \text{operator }<o>) \\
  & (<o> \text{ name execute-tactic} \\
  & \text{action.achieved *yes*})
\end{align*}
\]

\[
\begin{align*}
  & --> \\
  & (<o> \text{ achieved *yes*}) \\
  & }
\]

Execute-Tactic Termination

\[
\begin{align*}
sp \{ & \text{execute-tactic*terminate} \\
  & (\text{state }<s> \text{ operator }<o>) \\
  & (<o> \text{ name execute-tactic} \\
  & \text{achieved *yes*})
\end{align*}
\]

\[
\begin{align*}
  & --> \\
  & (<s> \text{ operator }<o> \emptyset) \\
  & }
\]
New Employ-Weapons Proposal

sp {intercept*suggest-proposal*employ-weapons
  (state <s> ^problem-space.name intercept
   ^threat-agent <a>)
  (<a> ^id.value bandit
   ^contact.achieved *yes*)
  ;## But not if we are about to pick a new primary threat or execute
  ;## an intercept tactic
  -(state <s> ^operator <o> +)
  (<o> ^name << select-primary-threat execute-tactic >>)
-->
  (<s> ^suggest-proposal <p> + &)
  (<p> ^name employ-weapons
   ^agent <a>)
}

Supporting Knowledge Structures

What knowledge structures support the execute-tactic operator?

- Record of tactic type
- Parameters for tactic
Supporting Knowledge Structures

In `top-ps/command-elaborations.soar`

```lisp
## Set up a structure telling us which tactical intercept action to use
sp {top-ps*elaborate*command*group*tactical*action
   (state <s> ^problem-space.name top-ps
    ^io.input-link.section.strategy.intercept <si>
    ^command.group <cg>)
   (<cg> ^type section
    ^tactical <gt>)
-->
   (<gt> ^action <tac>)
   (<tac> ^type <si>)
}

## Set up a parameter for when to initiate a pincer maneuver
sp {top-ps*elaborate*command*group*tactical*action*initiate-range
   (state <s> ^problem-space.name top-ps
    ^command.group.tactical.action <tac>)
   (<tac> ^type pincer)
-->
   (<tac> ^initiate-range 80000)
}
```

Execute-Tactic Elaborations

What local data structures do we need to execute a tactic?
Execute-Tactic Elaborations

What local data structures do we need to execute a tactic?

- Initialization
  - Record the new problem-space name, "execute-tactic"
- The target agent for the tactical maneuvers
- A pointer to the data structure for the tactic
- A pointer to the command group structure
- An indication that it is okay to start the maneuver
  - If lead, we must have communicated the maneuver
  - If wingman, just go ahead

Execute-Tactic Elaborations

In execute-tactic/elaborations.soar:

```
sp {execute-tactic*elaborate*state*problem-space
   (state <s> `superstate.operator.name execute-tactic
      `impasse no-change
      `attribute operator)
   -->
   (<s> `problem-space <p>)
   (<p> `name execute-tactic)
 }

sp {execute-tactic*elaborate*state*target-agent
   (state <s> `problem-space.name execute-tactic
      `superstate.operator.target-group.member <tgm>)
   (<tgm> `lead *yes*
      `agent <a>)
   -->
   (<s> `target-agent <a>)
 }
```
Execute-Tactic Elaborations (cont.)

sp {execute-tactic*elaborate*state*tactic
    (state <s> "problem-space.name execute-tactic
        "superstate.operator.action <sta>)
  -->
    (<s> "action <sta>)
}

sp {execute-tactic*elaborate*state*command-group
    (state <s> "problem-space.name execute-tactic
        "superstate.operator.command-group <cg>)
  -->
    (<s> "command-group <cg>)
}

sp {execute-tactic*elaborate*state*okay-to-start*lead
    (state <s> "problem-space.name execute-tactic
        "command-group.tactical.role lead
        "communicated *yes*)
  -->
    (<s> "okay-to-start *yes*)
}

sp {execute-tactic*elaborate*state*okay-to-start*wingman
    (state <s> "problem-space.name execute-tactic
        "command-group.tactical.role subordinate)
  -->
    (<s> "okay-to-start *yes*)
}
Execute-Tactic Sub-Operators

What are the sub-tasks in executing a particular tactic (pincer)?

- Communicate-Pincer
- Start-Pincer
- Slow-Pincer-Turn
Communicate-Pincer

The group lead must communicate the initiation of the tactic to the wingman

- Propose when the tactic has not yet been communicated
- Apply
  - Send radio message to begin pincer
  - Mark that tactic has been communicated
- Terminate when communication is complete

Communicate-Pincer Proposal

sp {execute-tactic*propose*communicate-pincer
   (state <s> ~problem-space.name execute-tactic
    ~command-group.tactical.role lead
    ~communicated *yes*
    ~action.type pincer)

   -->
   (<s> ~operator <o> + =)
   (<o> ~name communicate-pincer
    ~type output)
}
Communicate-Pincer Application

sp {communicate-pincer*apply
   (state <s> ^operator <o>
      ^top-state.call-sign <cs>
      ^command-group.mission.radio <radio>
      ^communicated *yes*)
   (<o> ^name communicate-pincer)
   -->
   (<s> ^communicated *yes*)
   (<o> ^desired-radio-comm <drc>)
   (<drc> ^radio <radio>
      ^message <mcs>)
   (<mcs> ^item <cs>
      ^next <mpince>)
   (<mpince> ^item pince)
}

Communicate-Pincer Termination

sp {communicate-pincer*terminate
   (state <s> ^operator <o>
      ^communicated *yes*)
   (<o> ^name communicate-pincer)
   -->
   (<s> ^operator <o> 0)
}
Start-Pincer

A pincer starts with a hard turn away from your partner

- Propose when it is okay to start the maneuver and the maneuver has not already been started
- Turn perpendicular to target, and away from partner
- Wait until desired heading is achieved, then
- Mark that pincer has been started
- Terminate when pincer is marked as started

Start-Pincer Proposal

sp {execute-tactic*propose*start-pincer
   (state <s> ~problem-space.name execute-tactic
      ~action.type pincer
      ~okay-to-start *yes*
      ~maneuver-started *yes*)
  -->
  (<s> ~operator <o> + =)
  (<o> ~name start-pincer
       ~type output
       ~requires wait)
}

Start-Pincer Application

Direction to turn depends on which side of the formation we are on

sp {start-pincer*apply*desired-heading*left
  (state <s> ~operator <o>
    ~command-group.formation.side left
    ~target-agent.relative.magnetic-bearing.value <mb>)
  (<o> ~name start-pincer
    ~desired-heading)
  -->
  (<o> ~desired-heading (round-off-heading (- <mb> 90) 1)
    ~desired-turn-rate hard)
}

Start-Pincer Application (cont.)

sp {start-pincer*apply*desired-heading*right
  (state <s> ~operator <o>
    ~command-group.formation.side right
    ~target-agent.relative.magnetic-bearing.value <mb>)
  (<o> ~name start-pincer
    ~desired-heading)
  -->
  (<o> ~desired-heading (round-off-heading (+ <mb> 90) 1)
    ~desired-turn-rate hard)
}

### Mark when the turn is complete
sp {start-pincer*apply*maneuver-started
  (state <s> ~operator <o>
    ~maneuver-started *yes*)
  (<o> ~name start-pincer
    ~achieved desired-heading)
  -->
  (<s> ~maneuver-started *yes*)
}
Start-Pincer Termination

sp {start-pincer*terminate
   (state <s> ~operator <o>
      ~maneuver-started *yes*)
   (<o> ~name start-pincer)
   -->
   (<s> ~operator <o> @)}

Slow-Pincer-Turn

After the initial cut, turn slowly back toward the target

- Propose when pincer has been started, but tactic is not achieved
- Make slow turn toward target
- Wait until contact with the target is achieved, or
- Until the desired heading is achieved, then
- Mark the tactic as achieved
- Terminate when the tactic is achieved
Slow-Pincer-Turn Proposal

sp {execute-tactic*propose*slow-pincer-turn
   (state <s> ^problem-space.name execute-tactic
      ^action.achieved *yes*
      ^maneuver-started *yes*)
   -->
   (<s> ^operator <o> + =)
   (<o> ^name slow-pincer-turn
      ^type output
      ^requires wait)
}

Slow-Pincer-Turn Application

### Turn back toward the target, but slowly
sp {slow-pincer-turn*apply*desired-heading
   (state <s> ^operator <o>
      ^target-agent.relative.magnetic-bearing.value <mb>)
   (<o> ^name slow-pincer-turn
      ^desired-heading)
   -->
   (<o> ^desired-heading <mb>
      ^desired-turn-rate standard)
}
Slow-Pincer-Turn Application

### When (if) we reacquire contact with the target, the tactic is done
sp {slow-pincer-turn*apply*achieved*contact
    (state <s> `operator.name slow-pincer-turn
        `action <sta>
        `target-agent.contact.achieved *yes*)
    (<sta> `achieved *yes*)
    -->
    (<sta> `achieved *yes*)
}

### Otherwise, just mark the tactic as achieved when we have completed
### the turn
sp {slow-pincer-turn*apply*achieved*desired-heading
    (state <s> `operator <o>
        `action <sta>)
    (<o> `name slow-pincer-turn
        `achieved desired-heading)
    (<sta> `achieved *yes*)
    -->
    (<sta> `achieved *yes*)
}

Slow-Pincer-Turn Proposal

sp {slow-pincer-turn*terminate
    (state <s> `operator <o>
        `action.achieved *yes*)
    (<o> `name slow-pincer-turn)
    -->
    (<s> `operator <o> @)

Other Concerns

- Wingman must understand and process the "pince" message
- Execute-tactic operator must also be proposed for wingman
  - In follow-leader problem space

Wingman's Message Processing

In applications/messages.soar

sp {execute-mission*apply*command*group*tactical*action*ordered-to-start
  (state <s> "operator.name execute-mission
      ^command.primary-group <pg>
      ^comm.message <m>)
  (<pg> "mission.radio <radio>
      ^tactical <gt>)
  (<gt> "role subordinate
      ^lead.call-sign <lead-cs>
      ^action <gta>)
  (<gta> "type pincer
      ^ordered-to-start *yes*)
  (<m> "content <mc>
      ^radio <radio>
      ^processed)
  (<mc> "item <lead-cs>
      ^next <mprise>)
  (<mprise> "item pince)
  -->
  (<m> "processed *yes*)
  (<gta> "ordered-to-start *yes*)
}
Wingman's Execute-Tactic Proposal

(sp {follow-leader*propose*execute-tactic
  (state <s> "problem-space.name follow-leader
    "top-state <ts>)
  (<ts> "command.primary-group <cg>
    "target.group <tg>)
  ;## Figure out which group we are intercepting
  (<tg> "member.agent.commit-criteria.satisfied *yes*)
  (<cg> "tactical.action <sta>)
  (<sta> "ordered-to-start *yes*
    "achieved *yes*)
  -->
  (<s> "operator <o> + , =)
  (<o> "name execute-tactic
    "action <sta>
    "command-group <cg>
    "target-group <tg>)
}

Wingman's New Fly-To-Formation Proposal

(sp {follow-leader*suggest-proposal*fly-to-formation
  (state <s> "problem-space.name follow-leader
    "command-group <cg>)
  (<cg> "formation.lateral-range <lr>
    "tactical.lead.agent <a>)
  -(<lr> "value much-too-far)
  -{(state <s> "operator <o> +)
    (<o> "name execute-tactic)}
  -->
  (<s> "suggest-proposal <sp> + &)
  (<sp> "name fly-to-formation
    "agent <a>)
}

Soar/IFOR Tutorial 4.4 and Exercise 4.4: Extending Agents, Part Three
Wingman’s New Achieve-Proximity Proposal

```lisp
sp {follow-leader*suggest-proposal*achieve-proximity
    (state <s> "problem-space.name follow-leader
        "command-group <cg>
    (<cg> "tactical.lead.agent <a>
        "formation.lateral-range.value much-too-far
    )-{(state <s> "operator <o> +)
        (<o> "name execute-tactic)}
    -->
    (<s> "suggest-proposal <p> + &)
    (<p> "name achieve-proximity
        "agent <a>)
}
```

---

Give It A Try

Run soarsf with the microtas exercise.

The agents in the main scenario have been told to do a pincer, but they did not know how before.

Let’s see what they do now.
Other Improvements

There are certainly many other improvements and extensions that could be made to the Micro-TacAir-Soar agents. Think about some you might like to add.

Examples:

- Get the wingman’s pincer maneuver to finish better
- Get the wingman to shoot when the opportunity arises
- Try implementing some other intercept tactic
- Get the agent to do an “F-Pole” after firing a missile

An Alternative Domain

A ship evading an incoming torpedo

- When should we propose the Evade operator?
- What problem space(s) should Evade belong to?
- What should the Evade operator do?
- Does the operator interact with the proposal of other operators?
- What local data structures do we need to carry out the evasive maneuvers?
- What are the sub-tasks involved in evading a torpedo?
- Are there any other concerns involved in implementing the new capability?
Talk 5.1: Adding Soar Control of Ships

Frank Vincent Koss

May 3, 1996, 9:00am - 9:30am

Soar/IFOR Tutorial
University of Michigan AI Lab

Scope

This talk will outline the changes that would need to be made to the SMI and ModSAF in order to allow Soar agents to control a ship. The following are assumed:

- Familiarity with C programming
- Familiarity with ModSAF conventions
- An existing hull library in ModSAF (e.g. libship)
- An existing .rdr file (e.g. US_CVN72_params.rdr)
- A working version of SoarSF

This will only outline the process. Parts of the process most likely will change by the time ships are supported by ModSAF.
Modifications to ModSAF

A line must be added to
common/src/ModSAF/entities/models.rdr. An entry is
needed to signify that the vehicle may be controlled by Soar.
Make a copy of the line for the vehicle of interest, e.g.

("vehicle_US_CVN72" US_CVN72_MODEL_PARAMETERS
SHIP_STD_PARAMS)

and change the copy to read:

("vehicle_US_CVN72-Soar" (SM_SAFSOAR)
US_CVN72_MODEL_PARAMETERS SHIP_STD_PARAMS)

New output commands

Caveat: I don’t know anything about piloting ships. I’m going to
make some assumptions to illustrate a process.

Two new output commands will be needed to control the ship,
since the engines and anchor are set when controlling a ship, not
the heading or speed.

The two new commands will be:

^desired-anchor integer up/down

^desired-engines float fraction of full

So, (^desired-anchor 0) means “lower anchor” and
(^desired-engines 1.0) means “all ahead full”.

Adding I/O symbols

All files from now on are found in common/libsrc/libsafoear

Each output command must have a corresponding I/O symbol. They are in the soar_modsaf_io data structure of each agent. This is defined in libsfr_local.h typedef of soar_modsaf_io_struct. The beginning looks like this:

```c
typedef struct soar_modsaf_io_struct {
    char * agent_name;
    agent * soar_agent;         /* Attached Soar agent */
    SAFSOAR_VARS * modsaf_vehicle;    /* ModSAF vehicle I/O info. */

    Tcl_Interp *interp;          /* This agent's Tcl interpreter */

    enum Agent_type soar_agent_type;

    Symbol *top_input_link_id;

    Symbol *break_lock_sym;
    Symbol *command_sym;
```

Adding I/O symbols, continued

At the end of the list of Symbol * definitions, add

```c
Symbol *desired_anchor_sym;
Symbol *desired_engines_sym;
```

These symbols must be created and freed. In the function make_soar_modsaf_io_structure lines such as:

```c
smio->break_lock_sym = get_io_sym_constant ("break-lock");
smio->command_sym = get_io_sym_constant ("command");
```

create I/O symbols. Add the following to the existing ones:

```c
smio->desired_anchor_sym = get_io_sym_constant ("desired-anchor");
smio->desired_engines_sym = get_io_sym_constant ("desired-engines");
```
Adding I/O symbols, continued II

I/O symbols are freed in `destroy_soar.modsaf.io.structure` with the following:

```c
release_io_symbol(smio->desired_anchor_sym);
release_io_symbol(smio->desired_engines_sym);
```

Processing the output command

Vehicle commands are processed in `send_change_flight_parameters`. The command must be read from the output link, the argument extracted, and the command sent to the hull. The following does all this.

```c
arg = get_output_value(outputs, output_link_id, smio->desired_anchor_sym);
if (arg)
  if (c_symbol_value(arg) == 0)
    HULLS_SET_ANCHOR(vehicle_id, hull, HULLS_LOWER_ANCHOR);
  else
    HULLS_SET_ANCHOR(vehicle_id, hull, HULLS_RAISE_ANCHOR);
arg = get_output_value(outputs, output_link_id, smio->desired_engines_sym);
if (arg)
  HULLS_SET_ENGINES(vehicle_id, hull, c_symbol_value_float(arg));
```

All of the apparently undefined variables are defined at the top of `send_change_flight_parameters` and reused throughout it.
Input link attributes

In order to sail, the agent needs to know the state of the anchor (up or down). This will come in on the input link as:

(K1 ... ^anchor <x7> ...)  
(<x7> ^state *up*)

This will be updated each cycle based on the state of the anchor, which will come from ModSAF.

Input C data structures

A new field needs to be added to the agent's data structure to access the anchor input object. This is added to the soar_modsaf_io_struct as before. A good place is just before the lines:

float64 old_alt; /* old_* fields are needed to compute */  
float64 old_daltitude; /* various derivatives */

A sim_obj (simulation object) must be added for the anchor input object:

sim_object *anchor_obj;
Input C data structures, continued

Since this is just a pointer, the data structure itself needs to be allocated. This is done in `ifor_input_function` when the agent is first created (the top state is created). In the section with lines such as:

```c
smio->battle_area_bin = make_sim_obj();
smio->debug_obj = make_sim_obj();
```

add

```c
smio->anchor_obj = make_sim_obj();
```

The object needs to be initialized. We’ll set it to `down` in this same part of the code with:

```c
setAVstring(smio->anchor_obj, "state", "*down*");
```

Adding to the input link

This needs to be added to the input link as soon as the agent is created. This is done in `safsoar.c` in the function `create_initial_input_structures` with the following calls:

```c
add_sim_object_to_input_link(smio->top_input_link_id, "anchor",
                           smio->anchor_obj);
internalize_world_object(smio->anchor_obj);
```
Updating the value

Each cycle, the anchor state must be read and this object updated. Anything that needs to be processed regularly is placed in \texttt{ifor\_input\_function} and is run during each input cycle. After:

\begin{verbatim}
if (smio->debug_changed) {
    update_debug_flag(smio);
}
\end{verbatim}

Add the following:

\begin{verbatim}
if (HULLS\_GET\_ANCHOR\_STATE(vehicle_id, hull) == HULLS\_ANCHOR\_UP)
    setAVstring(smio->anchor_obj, "state", "*up*")
else
    setAVstring(smio->anchor_obj, "state", "*down*");
internalize\_world\_object(smio->anchor_obj);
\end{verbatim}

\textbf{Note}: The above should be in a function, not in \texttt{ifor\_input\_function} itself. Also, there should be state associated with the anchor so that setAVstring and internalize\_world\_object are only called if the state of the anchor has changed. I've done it this way in this example for simplicity.

Removing the structure

The state of the anchor on the input link will change each input cycle as appropriate. When the agent is destroyed, the structures need to be freed. This is accomplished in \texttt{ifor\_input\_function}, when the top state is removed, after lines such as:

\begin{verbatim}
release\_io\_symbols\_simtree(smio->debug\_obj);
\end{verbatim}

with

\begin{verbatim}
release\_io\_symbols\_simtree(smio->anchor\_obj);
\end{verbatim}
Multiple agents per vehicle

- Now, an agent is created when a vehicle is created. The agent callsign is the marking of the vehicle.
- Proposal: when certain types of vehicles are created (carrier, cruiser, etc.), the vehicle marking will be used to find a file. His file will contain the names of the agents to be created on this vehicle.

Problems with multiple agents per vehicle

- Communication: the only interagent communication is through radios, which everyone can hear. Also, all agents have access to all radios. But then there's CCSIL...
- Control conflicts: an agent has full control of its vehicle. Multiple agents can attempt to, for example, steer the vehicle at the same time.
5.2: Knowledge Acquisition

Randolph M. Jones

May 3, 1996, 9:30AM–10:00AM

Soar/IFOR Tutorial

University of Michigan AI Lab

Acquiring the Knowledge for Intelligent Behavior

An important part of building intelligent agents involves gleaning knowledge from domain experts.

This talk briefly presents some of the concerns and methods we have identified in acquiring knowledge for TacAir-Soar.
Outline

- Start with a model for behavior
- Use model to constrain what types of knowledge you need to acquire
- Choose a good expert
- Decompose all tasks into goals and methods for achieving the goals
- Ask leading questions
- Use "spiral" development model for interleaving knowledge acquisition and system design

Use a Model for Behavior to Constrain Acquisition

- Provides a framework in which to fit expert knowledge
  - Domain can be divided into problem spaces
- Grain-size of behavior and concepts
  - Behaviors are step-wise operator application through problem spaces
  - Knowledge in fine-grained chunks...no monolithic scripts for behavior
  - Time-scale of behavior generation matches time-scale of human deliberate actions
- Constraints on how behavior must be modeled
  - Reasons matter...all decisions grounded in chains of reasoning to goals and perception
Choose a Good Expert

- Expert must be very knowledgeable of domain
  - Preferably with up-to-date knowledge
- Must understand and be able to convey reasons for decisions
  - Instructors are often a good choice
- Must be able to lay out space of behaviors in general as well as specific examples
- Must have good communication ability and be willing to help

Interviewing the Expert

- Start with a simple scenario as a skeleton
- Build up breadth and depth of scenario
- Record all details in many forms
  - Notes, graphical aids, drawings, videotape
- Generate formal summary of session quickly
Decompose Tasks into Goals and Methods

- Corresponds to how reasoning occurs in the selected model (Soar)
- It is not enough to stick to the level of "missions" or scenarios
  - But these provide good contexts for discovering individual goals and methods
- Flexible, compartmentalized units of knowledge
  - Supports reuse of goals, behaviors, and concepts
- Helps the knowledge engineer understand the domain

Ask Leading Questions

- Experts are not (usually) knowledge engineers
  - They do not know what you need to know, or at what level of detail
- Try to formalize an understanding of the current task, and get corrected if wrong
- Usually helps to phrase questions in the context of a specific scenario
- Then explore variations to the scenario
Use Spiral Development

- Start with a specific scenario
- Analyze scenario
  - Decision points
  - Potential goals
  - Behaviors
  - Potential reasons for behaviors
- Based on analysis, propose variations on initial scenario

Use Spiral Development (cont.)

- Confirm or modify analysis with new questions on proposed variations
- Slowly focus in on the concepts, goals, and behaviors from continued interaction with expert
- Implement code based on current formalization of the domain
  - Makes explicit functional constraints on behavior
  - Raises new questions to ask the expert
5.3: Creating New Soar/IFOR Agents

Paul E. Nielsen

May 3, 1996, 10:15-12:30

Soar/IFOR Tutorial

University of Michigan AI Lab

Overview

This talk addresses the problem of creating an entirely new agent from a Soar/IFOR perspective.

Creating new agents will still be based on existing code, but constitutes a much larger task than adding or modifying an operator or problem space.

Rather than a structured talk, most of this session will be working with you on the initial development of an entirely new IFOR agent.

Before you begin, you will need to have a ModSAF model of the agent's vehicle type and an Soar/ModSAF interface for this vehicle as described previously by Frank Koss.
Modify the Agent Definition Files

For Soar to recognize and interact with your new vehicle there are a number of files you’ll have to modify.

1. `top-ps/execute-mission/applications/vehicle.soar`

The Soar/IFOR code uses this file to determine if your vehicle is blue (friendly) or red (opponent).

Add your new vehicle type to the appropriate set of vehicles.

---

Modify the Agent Definition Files (continued)

2. `top-ps/vehicle/<vehicle-type>.soar`

This file contains most of the vehicle specific parameters.

For example `top-ps/vehicle/fa-18.soar` has:

- Fuel parameters (capacity, consumption, transfer rates, joker, and bingo)
- Maneuver parameters (max speed, attack speed, lowest desired speed, etc.)
- Missile information (lateral separation, target aspect, cuts, and evasion range)
- Transit parameters (normal transit speed, min/max speed, altitude, and turn rates)
- Radar modes
Modify the Agent Definition Files (continued)

There is also a production which says this vehicle is capable of both fighter and attack missions.

Once you've defined your agent's parameters, add your new file to `top-ps/vehicle/load.soar` so it gets loaded properly.

---

Modify the Agent Definition Files (continued)

3. `top-ps/vehicle/other.soar`

This file has a set of generic parameters for undefined agents.

Once you have defined a vehicle specific parameter file, add your vehicle type to the list of vehicles which do *not* need this production

*This is not part of Micro TAS.*
Modify the Agent Definition Files (continued)

4. top-ps/parameters.soar*

If your agent will fly, add it to the list of known airborne agents in top-ps*elaborate*parameter*known-airborne-agents

*This is not part of Micro TAS.
Modify the Agent Definition Files (continued)

5. top-ps/vehicle-elaborations.soar

Finally, top-ps*elaborate*state*vehicle*plane defines radar, radar warning, and weapon types for the vehicles.

It is used by Soar/IFOR agents to define what they know about their potential opponents.

Defining Your Agent

Now your agent has a vehicle.

We’ll skip the knowledge acquisition process.

Next you will define your agent’s mission.
Re-use Low Level Capabilities

By now you should be an accomplished web surfer and have a good feel for finding your way through our documentation.

Adapt the existing code as you see fit.

Creating an Entirely new IFOR Agent

The remainder of the tutorial will be an open programming exercise for creating your own IFOR agent.

If you don’t already have an agent in mind, you can use the unmanned air vehicle.