Monday, April 29th

Day 1: Intro, domain description, and exercise editor

8:30-10:00  Talk 1.1: Intro/Overview/Demo (John)
10:00-10:15 break
10:15-10:45 Talk 1.2: Domain description, part one (Randy)
  • levels of organization

10:45-11:30 Exercise 1.1: Find WWW domain description document (Paul)
11:30-12:30 Talk 1.3: Domain description, part two (Randy)
  • individual aircraft

12:30-1:30 lunch
1:30-2:00 Talk 1.4: Soar and ModSAF (Frank)
  • Running IFOR's with different scenarios
  • Talk merges with following exercises

2:30-3:00 Exercise 1.2: Run existing scenarios in ModSAF (Frank)
3:00-3:15 break
3:15-4:30 Talk 1.5: Using the exercise editor (Karen)
  • Editing scenarios and creating new scenarios
  • Talk merges with following exercises

4:30-5:30 Exercise 1.3: Add a mission to an existing scenario and run (John)
Tuesday, April 30th

Day 2: Domain, problem spaces, and intro to Soar

8:30-9:00 Recap of day 1, questions, etc. (Group)

9:00-9:45 Talk 2.1: Domain description, part three (John)
  • Groups of aircraft

9:45-10:15 Talk 2.2: Domain description, part four (Paul)
  • Groups of aircraft interacting with controllers

10:15-10:30 break

10:30-11:30 Exercise 2.1: More work with entering missions and running (John)
  • Also, running Soar/ModSAF over the network

11:30-12:00 Talk 2.3: Overview of the problem spaces in TacAir-Soar (John)

12:00-1:00 lunch

1:00-2:45 Talk 2.4: Intro to Soar (Clare)
  • (merges with following exercises)

2:45-3:15 Exercise 2.2: Running Soar with simple blocks-world (Clare)

3:15-3:30 break

3:30-4:00 Talk 2.5: Soar Syntax (Clare)

4:00-4:30 Talk 2.6: Soar Operators: Selection and Application (Peter)

4:30-5:30 Exercise 2.3: Blocks world with separate pick-up and put-down (Peter)
**Wednesday, May 1st**

**Day 3: Soar syntax, TAS operators**

<table>
<thead>
<tr>
<th>Time</th>
<th>Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>8:30-9:00</td>
<td>Recap of day 2, questions, etc. (Group)</td>
</tr>
<tr>
<td>9:00-9:30</td>
<td>Talk 3.1: Example operators in TacAir Code (Randy)</td>
</tr>
<tr>
<td>9:30-10:00</td>
<td>Talk 3.2: Soar Impasses and Subgoals (Peter)</td>
</tr>
<tr>
<td>10:00-10:15</td>
<td>break</td>
</tr>
<tr>
<td>10:15-10:30</td>
<td>Talk 3.3: MicroTAS (Randy)</td>
</tr>
<tr>
<td>10:30-11:15</td>
<td>Exercises 3.1: From problem space level to operators (Randy)</td>
</tr>
<tr>
<td>11:15-11:30</td>
<td>Talk 3.4: Persistence (Clare)</td>
</tr>
<tr>
<td>11:30-12:30</td>
<td>Exercise 3.2: From operator level to productions (Randy)</td>
</tr>
<tr>
<td>12:30-1:30</td>
<td>lunch</td>
</tr>
<tr>
<td>1:30-2:00</td>
<td>Talk 3.5: Soar I/O (Clare)</td>
</tr>
<tr>
<td></td>
<td>-- also cover inputs and outputs from ModSAGE (Clare)</td>
</tr>
<tr>
<td>2:00-3:00</td>
<td>Exercise 3.3: Running and debugging Soar (Clare)</td>
</tr>
<tr>
<td>3:00-3:30</td>
<td>Talk 3.6: SDE (John)</td>
</tr>
<tr>
<td>3:30-3:45</td>
<td>break</td>
</tr>
<tr>
<td>3:45-5:30</td>
<td>Exercises 3.4: More with Blocks-World (Peter)</td>
</tr>
<tr>
<td></td>
<td>-- subgoaling</td>
</tr>
</tbody>
</table>
Soar History: 1981

First Soar developed as John Laird’s thesis research at Carnegie Mellon University.

1. Software architecture to support problem solving with many different methods.
2. Integrated rule-based systems with problem spaces.
4. Demonstrated that a single architecture could support many different methods for many different tasks: Generate and Test, Depth-first Search, Breadth-First Search, Means-ends Analysis, Hill Climbing, ...

Soar History: 1981-1987

Continued development, evolution, and application

1. 1982: Added subgoals to provide multi-level problem solving.
2. 1984: Added learning (called chunking from Paul Rosenbloom’s thesis) as integral part to Soar.
Soar History: Modeling Human Behavior

1. Immediate reasoning tasks
2. Cryptarithmetic
3. Tower of Hanoi
4. Transcription Typing
5. Instructions for simple tasks
6. Verbal learning
7. Series completion
8. Many language processing phenomena
9. Concept learning
10. Visual attention
11. Dual tasks

Soar History: Example Applications

• Medical diagnosis (NeoMycin-Soar, Red-Soar)
• Production line scheduling (Merl-Soar)
• Algorithm design (Designer-Soar, Cypress-Soar)
• Chemical process modeling
• Natural language understanding (NL-Soar)
• Intelligent Tutor (ET-Soar)
Soar History: 1988-

Expand Soar’s ability to interact with external environments. Interfaced Soar to robotic and simulation systems. Greatly improve Soar’s efficiency.

- Lego-robot
- Robo-Soar: Puma-arm with TV camera
- Hero-Soar: Small mobile robot control.
- Air-Soar: Fly a simulated airplane with stick-level control.
- Rewrite Soar in C: 15-20 times speed up.
- Continued application in research on language, learning, modeling humans, ...
- Soar workshop once a year (15th at USC/ISI in June).

Soar History: Summary

- Progressed from a problem solving architecture to a complete architecture for building intelligent systems: reacting, planning, learning, interacting with the world, ...
- Used for both hard-core cognitive modeling and hard-core AI tasks.
Soar History: WISSARD/IFOR Project

- Attended first Computer Generated Forces and Behavior Representation Conference and met Dennis McBride (ARPA/ASTO).
- Agreed to work on Navy Air as part of WISSARD.
- Goal was to develop human-like computer forces for interactive simulation. Had planned on doing learning and automated knowledge acquisition.
- Started on 1v1.
- First users of ModSAF from BBN.
- Participated in STOW-E, ED-1.
- Now funded under CFOR through NRAD.

SoarSF + Exercise Editor
Soar/IFOR Philosophy

- Entity-level simulation of all vehicles:
  no aggregation of small units.

- Model human doctrine and tactics at the second to second level:
  Everything above eye-movements and hand movements.

- Completely autonomous agents:
  No human intervention except through simulated radio messages.
  Agents can plan if they have to.
  Agents can communicate with other agents as appropriate.

- All agents at the level of CFOR agents.

Soar/IFOR FWA: ED-1, Event 1

Missions:
BARCAP:
  luke 1&2 @ Enterprise
  skywalk 1&2 @ Kirk
Tanker:
  mobil-1 @ Uhuru
Controllers:
  E2C: dstar-1@Romulan

Capabilities:
1. Inflight Refueling
2. Simulated Launch
3. 2v2 BVR
4. E2C Control
   a. Assign to CAP points
   b. Bogey Status
   c. Weapons Status
5. Exercise Control
   a. Direct fighters

San Clemente

Uhuru

mobil-1

Luke-1

Enterprise

Luke-2

Carl Vinson

Romulan

dstar-1

Kirk

October 19-20, 1995
Soar/IFOR FWA: ED-1, Event 2

Missions:
- Close-Air Support: Leia
  - Carl Vinson, San Clemente, Jaguar, Pepsi
  - Wanda, tanks, Root Beer, Jaguar
  - FAC(A); Yoda-1, search for tanks
- Tanker: Shell-1 @ Jaguar

Controllers:
- TACC: Endor-1
- TAD: Obi-1

Capabilities:
1. Inflight Refueling
2. Dynamic targeting
3. Attack Planning
4. Division-level Attack
   a. Split Sections
   b. PGMs
5. Multiple Controllers
   a. Check-in
   b. Prepared for 9-line
   c. Handoff
6. Coordination with FAC
   a. FAC find target
   b. FAC marks target
   c. FAC clears hot

October 19-20, 1995

Soar/IFOR Tutorial

1.1: History of Soar/IFOR Project

Soar/IFOR FWA: ED-1, Event 3

Missions:
- MiG Sweep: hans 1&2
  - Carl Vinson, San Clemente, Federation, Wanda, Rootbeer, San Clemente
- SEAD SA-9: chewie 1&2
  - Carl Vinson, San Clemente, Federation, Wanda, SA-9, Jaguar
- Strike Bridge-2: solo 1&2
  - Carl Vinson, San Clemente, Federation, Wanda, Bridge-2, Jaguar

Capabilities:
1. Multiple missions
2. Attack planning
3. Dynamic targeting
4. HARM missiles
5. Dumb bombs
6. Disrupt BARCAP
Soar History: Soar/IFOR

Currently split development across three sites:

- Fixed wing at University of Michigan.
- Rotary wing at Information Sciences Institute, USC.
- Natural language at Carnegie Mellon University.

Soar History: Current Efforts

- FWA: All Navy and Air Force missions for STOW-97: Defensive Counter Air (DCA), Offensive Counter Air (OCA), Close Air Support (CAS), Strategic Attack, Suppression of Enemy Air Defensive (SEAD), Interdiction. Current preparing for ED-2, SSIT-1 (July 8–12, 1996)
- CCSIL: Will be used for all communication by end of summer.
Soar Future

- Continue basic research in modeling humans and developing AI systems.
- Improve behavioral fidelity of IFOR.
- Apply technology to new domains: education and entertainment.

Soar Tutorial: Major Topics

- Connections between Soar and ModSF (SoarSF).
  Day 1 & 5
- Editing and creating new exercises with Exercise Editor.
  Day 1 & 2
- Basics of Tactical Air domain.
  Day 1 & 2
- Soar
  Day 2 & 3
- Micro-TacAir-Soar & TacAir-Soar
  Day 3, & 4
- Creating new Soar/IFOR agents
  Day 5

We hope you come away with an understanding of Soar and TacAir-Soar sufficient to read code. You probably won’t be able to code new systems on your own, but we are glad to have visitors.
1.2: Domain Description, Part One
Levels of Organization

Randolph M. Jones
April 29, 1996, 10:15AM–10:45AM

Levels of Organization

This talk presents a general overview of the Navy tactical air combat domain. The organization presented here matches the organization we have developed in or Domain Description Document on the World-Wide Web (http://ai.eecs.umich.edu/ifor/DDD).

This organization describes the tactical air domain as three interconnected levels of description.
Levels of Organization (cont.)

- General domain description
  - Description expressed in domain-specific language
  - General topics are aggregate behaviors and high-level, complex concepts
  - Best description for domain experts
- Functional decomposition
  - Break domain into intermediate and elementary concepts
  - Basic behaviors and capabilities that compose into aggregate behaviors
  - Best description for domain novices (?)

Levels of Organization (cont.)

- Computational implementation
  - Specify precise conditions for decision-making/behavior generation
  - Detailed descriptions of data structures
  - Best description for generating automated behavior

The first two levels are best for an introduction to the domain.
The third level is what the construction of synthetic forces aims for.
Tactical Air Combat: General Domain Description

There are three broad categories of information that describe the domain at the general level:

- Tasks (goals)
- Capabilities (behaviors)
- Knowledge ("data") structures

Tasks and Goals in Tactical Air Combat

"Tasks" are the goals and subgoals that provide direction for behavior.

There are many different levels of tasks and goals.

The highest levels include specific missions, which are actually collections of lower level tasks, goals, and constraints.

The subcomponents of specific missions include tasks and goals that are generally shared among many different missions.

These subcomponents represent the highest levels of the functional decomposition of the domain.
Missions

Examples of high-level missions for aircraft in the US Navy:

- Air-to-air missions
  - Sweep
  - BarCAP (Barrier Combat Air Patrol)
  - TarCAP (Target Combat Air Patrol)
- Air-to-ground/surface missions
  - CAS (Close Air Support)
  - Strategic attack
  - Interdiction
  - SEAD (Suppression of Enemy Air Defense)

Missions (cont.)

- Fixed-Wing Aircraft (FWA) support missions
  - Tanking
  - Reconnaissance
  - AEW (Airborne Early Warning)
  - Control
Common Capabilities

General capabilities are the classes of behaviors and actions that an agent can generate in the environment.

Behaviors are generated in response to existing tasks and the state of the environment.
Common Capabilities (cont.)

Examples of general tasks and capabilities that are shared between missions:

- Communication
- Coordination
- Control
- Intercept
- Weapons employment
- Classifying threats
- Flying in formation
- Specific tactics

Knowledge Structures

Knowledge structures are the internal representations that we hypothesize are "inside a pilot's head" (at some level of description). In a computational system, knowledge structures are realized as specific data structures.

Knowledge structures...

- specify the language and concepts with which the agent reasons
- represent domain information in various (possibly redundant) useful ways
- reflect the agent's perception of the true state of the environment
Knowledge Structures (cont.)

NOTE: It is very difficult to start talking about knowledge structures without moving toward the implementation level.

We cannot actually look inside Navy pilots' heads, but these structures correspond to an approximate knowledge organization that they seem to be using (based on our functional decomposition and analysis of the domain).

Knowledge Structures (cont.)

Examples of some of the knowledge structures that support Navy Air tasks and capabilities:

- Mental agents
- Command groups
- Control groups
- Bogey groups
- Ordnance descriptions
- Vehicle descriptions
- Sensor descriptions
- Formations
Exercise 1.1: Soar/IFOR Documentation on the Web

Paul E. Nielsen

April 29, 1996, 10:45-11:30am

Soar/IFOR Tutorial

University of Michigan AI Lab

Soar/IFOR Documentation

The vast majority of Soar/IFOR documentation is on-line and may be accessed over the World-Wide Web.

This exercise gives you a chance to acquaint yourself with the available documentation and quickly find what you need.

To get started login as ifor and type netscape at the prompt.
Soar/IFOR Documentation

The root of online documentation for Soar/IFOR project is at:
http://ai.eecs.umich.edu/ifor/

This contains links to many different subtrees, divided into:

- Introduction
- Primary Soar/IFOR Documentation
- Overview Poster
- Major Demonstrations and Events
- Additional Documentation
- Papers
- Other related sources of information

Introductory Documentation

The introductory material simply states

- Who we are and generally what we are doing
- The institutions working on this project
- The primary tools we are using
  - Soar
  - ModSAF
Primary Soar/IFOR Documentation

The primary documentation contains what we consider the core documentation on the Soar/IFOR project.

- Requirements for Automated FWA Behaviors*
- The Domain Description Document (DDD)*
- Functional Assumptions
- TacAir-Soar Implementation Documentation*
- Interface Documentation
- Documentation Search

*You will need a password to read these entries.

Requirements for Automated FWA Behavior

This document lays out all of the behaviors an automated FWA should be expected to demonstrate. For example under "AIR-TO-AIR BEHAVIORS" there is the "BASIC FLIGHT PROFILE" which contains:

1. Mission Planning
2. Mission Briefing
3. Prelaunch /Pre-Takeoff Activities
4. Aircraft Launch/Takeoff Activities
5. Departure Activities
6. Enroute Activities
7. Tactical Mission Activities
8. Enroute/RTB/RTF Activities
9. Recovery / Approach Activities
10. Landing Activities
11. Postflight Activities
12. Mission Debrief Activities
The Domain Description Document (DDD)
http://ai.eecs.umich.edu/ifor/DDD/

The DDD contains information we have extracted in knowledge acquisition sessions with our experts including the following:

- Communication
- Controllers
- Formations
- Glossaries
- Missions
- On-line manuals
- Scenarios
- Tactics
- Unincorporated KA notes

Throughout this document, there are links to our specific Soar/IFOR implementations.

---

TacAir-Soar Implementation Documentation
http://ai.eecs.umich.edu/ifor/DDD/tacair/

Contains the documentation which links into the code itself.

The code is organized according to problem spaces, which we’ll cover later, but for now you can think of them as subroutines.

Most of the links are for operators and follow a standard form.

- An overview of the operator
- The conditions for proposing this operator
- What the operator does
- The conditions for exiting the operator
- The actual Soar code which implements this operator
Interface Documentation

http://ai.eecs.umich.edu/ifor/interface/

Contains information about our interface between Soar and ModSAF.

- Input link
  - describes inputs Soar gets from ModSAF
- Output link
  - describes the outputs Soar sends to ModSAF
- Right-hand side functions
  - describes C functions which may be called from Soar

The Soar/IFOR WWW Documentation Search

http://ai.eecs.umich.edu/ifor/ifor-search.html

Keyword based search engine for finding relevant information quickly

Your search can include

- Domain Description Document (DDD)
- Implementation Documentation
- Soar/ModSAF Interface Documentation
- Soar Source Code
- All of the above

You can use logical connectors such as:
(dope and bogey) or (call and help)
The Relevance Rank

The relevance rank is a number which is generated with each result.

It is the program's "best guess" as to how relevant it thinks the file is to your query.

This rank number can range from 1 to 1000.

It depends on a number of factors:

- How many times your search word appears in the file
- How many words are in the file
- If the word appears in a title or header tag
- Other factors
Search Errors

**err: no results**
- Diagnosis: There were no results of the search.
- Solution: Try a different search.

**err: could not open index file**
- Diagnosis: Either the index file could not be found or it couldn’t be opened.
- Solution: The index file must be rebuilt.

**err: no search words specified**
- Diagnosis: No words were specified for searching.
- Solution: Fill in the “Search for” field of the search form.

**err: a word is too common**
- Diagnosis: A search word was used that was too common to give any meaningful feedback.
- Solution: Omit the common word and search again.

---

Scavenger Hunt

The purpose of the rest of the exercise is to familiarize you with finding the information you need in our documentation.

Each team will be given a list of things which be found somewhere in our documentation.

The winner will be the team which correctly answers the most questions before the end of this exercise.

In the event of a tie, the team finishing first will be declared the winner.

All decisions of the judges are final.

- This is a competition.
- There will be prizes.
Scavenger Questions

1. What is the link which says which way the pilot is looking?
2. How many degrees do we turn for an f-pole maneuver?
3. How do we select the altitude to begin an air to air engagement?
4. How many pages are in the first overview paper of the Soar/IFOR project?
5. Who or what is Judy? (Give a definition.)
6. What is the 9th line of a close air support mission brief?
7. What types of aircraft can launch HARM missiles?
8. When should a tac turn be used?
9. What information is “critical” when defending against a pincer?
10. How many types of multi-plane turns are used in Navy air?
11. What are the units for sine and cosine in Soar’s right hand side functions?
12. What message do you get when the documentation is missing?
1.3: Domain Description, Part Two
Individual Aircraft

Randolph M. Jones

April 29, 2996, 11:30AM–12:30PM

Soar/IFOR Tutorial

University of Michigan AI Lab

Capabilities Required for Individual Aircraft

This presentation describes the core capabilities for any individual agent that flies an aircraft in the Navy Tactical Air domain.

These capabilities allow agents to perform any of the sub-tasks required for each individual mission, but without the ability to coordinate, communicate, fit into a command hierarchy, etc.
Capabilities Required for Individual Aircraft (cont.)

We divide the discussion into four areas:

- **Sensors**
  - **Effectors**
    - Sensors and effectors are specific to a single vehicle
    - Thus, tasks specifically involved with sensors and effectors must be addressed at the individual level
  - **General mission sub-tasks**
    - All of the high-level common capabilities that individual agents must have in order to carry out all of the missions
  - **Data structures**
    - The knowledge and representations that support individual capabilities

1.3: Domain Description, Part Two  Individual Aircraft

Sensors

Every agent must be able to interpret information coming in on the sensors available in the aircraft. Currently, there are sensors that convey the following categories of information:

- **Vehicle status**
  - altitude, speed, heading, call-sign, fuel, world-time, ...
- **Radar blips**
  - altitude, speed, heading, collision-course, slant-range, ...
- **Visual objects**
  - similar to radar
- **Radio messages**
Sensors (cont.)

- Waypoint information
  - id, bearing, range, arrival-time, ...
- Radar warning information
  - bearing, level, type
- Weapons information
  - name, count
- CCIP (Computer Calculated Impact Point)
  - x, y, target-x, target-y, near, far, left, right
- Detonations
  - munition, result, time, x, y, z

Effectors

Effectors are the only interface through which the agent can have an influence on the world.

Some effectors lead (directly or indirectly) to a change in the environment. Some effectors involve manipulating the sensors in order to receive data about the environment. Thus, all effectors allow the agent in some way to influence its representation of the current state of the environment.

Every agent must have the knowledge of how to invoke each effector that is available in the existing aircraft.
Effectors (cont.)

The current set of effectors fall into the following categories:

- Vehicle maneuvers
  - desired-altitude, desired-heading, desired-speed, desired-turn-rate, ...
- Radar
  - desired-radar-mode, desired-radar-elevation, break-lock, ...
- Radio
- Waypoint computer
  - x, y, id, clear-waypoint-computer
- CCIP
  - x, y, disable-ccip

Effectors (cont.)

- Weapons selection and employment
  - select-missile, fire-missile, release-bomb, bomb-impact-separation, ...
General Tasks for Individuals

These are the "meat and potatoes" for agents in the Navy Tactical Air domain. Basic capabilities for achieving all of the different types of missions provide the structure on which further intelligent behavior can be built.

There are three very general required capabilities in this domain:

- Fly the aircraft
- Maintain situational awareness
- Employ weapons

General Tasks for Individuals (cont.)

In addition, there is a fourth category of capabilities that are specific to different types of missions:

- Achieve specific mission components

The following slides elaborate on the requirements of these four types of capabilities.
Flying the Aircraft

Naturally, most of the agents in this domain fly fixed-wing aircraft. Thus, keeping the aircraft airborne and making it go in the right place is usually the overriding concern.

Capabilities for flying the aircraft fall into three basic categories, from very simple to more complex:

- Individual maneuvers
- Navigation and route following
- Specific tactical maneuvers

Individual Maneuvers

TacAir-Soar flies aircraft with commands like desired-heading, desired-speed, desired-altitude, and their first derivatives.

ModSAF has flight dynamics to figure out how to achieve the desired values (not always possible, or can lead to stalling).

This is not fine enough control for many maneuvers:

- Close-range air-to-air combat
- Air-to-ground ordnance delivery

Together with LADS, we are developing an interface to ModSAF airframes that will allow finer control when necessary.
Navigation and Route Following

- Fly to waypoints (waypoint computer)
- Pre-specified flight paths
- Dynamic adjustments when necessary
- Control points
- Control routes
- Communication with controllers

Specific Tactical Maneuvers

- Evasive maneuvers
- F-Pole
- Intercept tactics
  - Pincer
  - Half-Pincer
  - Post-Hole
Maintaining Situational Awareness

This is an important capability for probably any intelligent agent in any domain you can think of. This is basically the capability of creating and maintaining an up-to-date and useful representation of the environment.

The agent’s representation of the current situation is crucial to determining which behaviors the agent will generate, and so this capability will have a direct impact on the quality of those behaviors.

Maintaining Situational Awareness (cont.)

In the Navy Tactical Air domain, there are a number of specific concerns for maintaining situational awareness:

- Existence of other agents
- Identification of threats
- Spatial representation of the environment
- Appropriate use of vision, radar, radar warning, and radio to paint the picture
Employing Weapons

Many of the agents in this domain must employ weapons of various types. Weapons are fired from the individual agent's vehicle, so much of employing weapons involves only individual capabilities. Following are some of the basic capabilities involved in employing weapons from an aircraft:

- Selecting a target
- Maneuvering target into weapons envelope
- Coordinating sensors with weapon
- Maneuver for best weapon shot
- Firing weapon
- Supporting weapon after firing
- Damage assessment

Achieving Specific Mission Components

In general, the Navy Air missions are combinations of a small number of specific pieces. In order to be capable of carrying out any of the high-level missions, an agent must have the ability to execute any of these components. This is a list of the mission components we have identified in the domain so far:

- CAP (Combat Air Patrol)
  - Fly racetrack
  - Reconnaissance
- Sweep
- Strike
- Intercept
- Deliver air-to-ground ordnance
Data Structures

Given the core capabilities required for an individual agent, we must also address the data structures that support these capabilities.

The following slides show examples of some of the knowledge structures that support Navy Air tasks and capabilities.

Data Structures (cont.)

- Flying
  - Vehicle parameters
    - vehicle
      - plane
        - name, type, radar, radar-warning, weapon
        - parameters
  - Waypoints
    - waypoint
      - point
        - id, x, y
        - computer
  - Flight plans/routes
    - flight-plan
      - leg
        - start, end
  - Tactical parameters
• Situation representation
  – Mental agents
    ^agent
    ^state
    ^call-sign, ^position, ^side, ^radar
    ^id
    ^relative
    ^my-position, ^lateral-range,
    ^magnetic-bearing, ^collision-course
    ^info
    ^source, ^radar, ^visual, ^comm, ^recorded
    ^contact
    ^achieved, ^active
    ^roe
    ^commit-criteria
    ^threat

– Information sources
  ^radar
  ^object
  ^type
  ^mode
    ^name, ^type, ^range, ^beam, ^notch
  ^mode
  ^blip
  ^vision
  ^object
  ^comm
    ^message
    ^content, ^radio

– Qualitative situation descriptions
Data Structures (cont.)

• Weapons information
  – Static weapons parameters
    ^weapon
    ^missile
    ^parameter
    ^type, ^name, ^range, ^guidance, ^speed, ^launch-to-eject
    ^lar
    ^quarter, ^slant-range-low, ^slant-range-high
  – Dynamic weapons envelope information
    ^weapon
    ^missile
    ^quarter
    ^lar-achieved, ^lar-reason
  – Active (fired) weapons
    ^weapon
    ^missile
    ^instance
    ^time-of-flight, ^in-flight, ^needs-support

Data Structures (cont.)

• Mission parameters
  – Mission type/objectives
    ^mission
    ^type, ^status, ^risk-type, ^scramble,
    ^takeoff-time, ^refueling-time
    ^cap
    ^type, ^orientation, ^length, ^altitude, ^duration
    ^transit
    ^speed, ^radar-mode, ^turn-rate, ^altitude
  – Subtasks
Talk 1.4: Introduction to SoarSF

Frank Vincent Koss

April 29, 1996, 1:30pm - 2:00pm

Soar/IFOR Tutorial

University of Michigan AI Lab

The SoarSF Process

UNIX Workstation

SoarSF Process

Soar  Soar-ModSAF Interface (SMI)  ModSAF

Network

ModSAF is the simulator written by Loral ADS. Soar is an artificial intelligence system written by a consortium of universities. SoarSF is the single program that is a combination of ModSAF and Soar.
The SoarSF Process, continued

SoarSF is a single process. Soar and ModSAF are compiled together to form SoarSF. The SMI mediates communication between Soar and ModSAF. In ModSAF, tasks and taskframes control vehicles. In SoarSF, either a Soar agent or tasks can control a vehicle. Multiple agents can control vehicles in one SoarSF process.

All communication, both among agents and among simulators, occurs through ModSAF.

- Communication among agents occurs through the ModSAF radio (over the network if necessary)
- Communication among simulators occurs through DIS and Persistent Object Database PDUs (over the network)

ModSAF versus Soar

ModSAF's scheduler runs the show.

Once each tick, once for each agent/vehicle pair, ModSAF passes control to Soar.

1. The SMI gets new inputs for the agent
2. Soar allows the agent one decision
3. The SMI sends the agent's output commands to ModSAF
The Soar-ModSAF Interface (SMI)

The Soar-ModSAF Interface (SMI) mediates between Soar and ModSAF.

- The SMI translates commands from the agent into the appropriate calls to ModSAF libraries.
- Sensory and status input from ModSAF is translated into a format understandable by Soar agents by the SMI.

The SMI is only invoked for Soar agents.

ModSAF

- Since most of you have used ModSAF, use of ModSAF itself will not be covered in depth.
- The exercise notes will have some reference material on the basics of using ModSAF.
- This will concentrate on the agent windows since they are specific to SoarSF.
Creating a Soar agent

A Soar agent must have a vehicle associated with it. At this time, there may only be one Soar agent per vehicle.

Vehicles to be controlled by a Soar agent are created in the same way as regular vehicles, except that the Methodology is set to Soar.

Only certain vehicles can be inhabited by Soar agents since changes need to be made to the definitions of some vehicles.

Logging In

- User: iforclas Password:
- cd TAS
- soarsf -ex tutorial -ev event

For the first exercise, we'll use tutorial for the exercise and 1 for the event.
Directory Structure

Each exercise has its own subdirectory, eg: tutorial, and in that subdirectory are the exercise.dat file (which has all the data for the exercise) and subdirectories for each event in the exercise Event1, Event2...

The Event subdirectories each have a scenarios/ subdirectory with the file scenario.dat, (the file loaded by SoarSF), and subdirectories for each agent participating in the event, agent-1/ agent-2/ ...

Directory Structure, continued

The following structure is what would exist for an exercise called tutorial that has one Event with two agents in that event, dstar-1 and luke-1:

```
-ifo/modsaf/
  soarsf
  exercises/
    tutorial/
      exercise.dat
      Event1/
      dstar-1/
      luke-1/
      scenarios/
        scenario.dat
    ...
    terrain/
      a2-ocean.c4b
```

*NOTE*: This example is not repeated accurately the format here is to make clear the organization of the files.
An agent window

Each Soar agent has its own window.

An agent window, continued

**File** menu: used for sourcing new files of productions.

**Show** menu: used to display different aspects of the agent’s environment.

**Help** menu: help on Soar and Tcl/Tk.

**Run** button: runs SoarSF

**Stop** button: stops SoarSF

**Print Stack** button: prints the current goal stack

**Simulation Speed** slider: controls simulation rate
Exercise 1.2: Running a Simulation

Frank Vincent Koss

April 29, 1996, 2:00pm - 3:00 pm

Soar/IFOR Tutorial

University of Michigan AI Lab

Running a Simulation

In this exercise, you will log in, run SoarSF, load a scenario, and manipulate the agents. This is to allow you to familiarize yourself with how the agents and the simulator interact.

Familiarity with ModSAF is assumed. If you are unfamiliar with it, the a few pages will explain the main concepts. You may also ask questions during the exercise.
The Event

Soar/IFOR FWA: ED-1, Event 1

Missions:
- BARCAP:
  - Luke 1&2 @ Enterprise
  - Skywalk 1&2 @ Kirk
- Tanker:
  - Mobii-1 @ Uhuru
- Controllers:
  - E2C: Dstar-1 @ Romulan

Capabilities:
1. Inflight Refueling
2. Simulated Launch
3. 2v2 BVR
4. E2C Control
   a. Assign to CAP points
   b. Bogey Status
   c. Weapons Status
5. Exercise Control
   a. Direct fighters

The Event Described

The event you will work with uses the full TacAir-Soar system.

There are six vehicles involved:


- **Skywalk-1, Skywalk-2**: two F/A-18's in a two-ship formation flying a DCA mission at Kirk.

- **Dstar-1**: an E-2C for Airborne Early Warning (AEW) on station at Romulan.

- **Mobil-1**: a KC-10A for in-flight refueling on station at Uhuru.

The Carl Vinson is just a marker, not a vehicle.
ModSAF

When SoarSF is run, a single large window will fill the screen. There are three parts to this window.

1. The menu bar runs along the top of the window and has a clock at the right end.
2. The tool bar runs along the left side of the window.
3. The majority of the window, with a grid in it, is the Plan View Display or PVD. It is in this area that you will see the vehicles as they move.

The Menubar

The most important menu for us is the File menu. The primary items are listed below.

- **Load Scenario...** allows you to load a scenario, a collection of vehicles and map objects. Each vehicle will have a mission when it loads.
- **New Scenario...** allows you to delete all of the current vehicles and map objects so that you can then reload the current scenario or load a different scenario.
- **Quit** quits SoarSF.

Sometimes the PVD will shrink. The bottom half of the window then becomes an editor. When done with the editor, the area becomes blank. To restore the PVD to the full size of the window, use the Special menu's **Show Editor** item.
The Toolbar

Some of the tools are described below. To leave a tool without it taking effect, click on the *Abort* button. If you wish the tool to take effect, click on the *Done* button after using the tool or editor.

- The blue tank is the **Unit Editor**. It creates an editor that allows you to create new vehicles.
- The top left red tool has the **User Preferences**. This allows you to change the units of values shown on the PVD. The parts labeled *Speeds* and *Altitudes* are the most important.

The Toolbar, continued

- The red "spider web" has the **PVD Controls**. It allows you to change some aspects of the PVD. The part labeled *Map Notations* allows you to add the Call Sign, Altitude, and Speed to the vehicles on the PVD.
- The red scissors is the **Delete Tool**. To remove a vehicle or map object from the PVD, click on it while in the Delete Tool. A red X will appear over the object. Multiple objects can be marked. Click on an object again to remove the mark. All marked objects will be deleted when the *Done* button is clicked on.
The PVD

The only things to do on the PVD are to zoom in and out.

As the green text at the bottom of the PVD says, click middle to zoom in. The point on which you click will be the center of the screen after zooming.

Click right to zoom out. Again, the point on which you click will be the center of the screen after zooming.

To zoom in on a particular area, click and drag with the middle button. The area you select will be enlarged to fill the PVD.

Running SoarSF

If you haven't already, log into the workstation. The username is iforclas, and the password will be given verbally.

Execute cd TAS to get to the directory that has SoarSF.

Run SoarSF by executing soarsf -ex tutorial -ev 1

The usual ModSAF messages will appear, followed by the ModSAF window.
Loading the Scenario

Load the scenario named `scenario.dat`.

The usual ModSAF dialog boxes will come up, and six agent windows will also appear. (The agent windows will be hidden behind the main window when the “Scenario has loaded” dialog box comes up.)

Each agent has a separate window. The title of the window and the prompt are both the name of the agent.

You may wish to turn on call signs in the PVD in order to see which window matches which vehicle.

Basic operations

In order to start the simulation running, type `schedule` in any agent window. This will start SoarSF running, which means the simulator will give Soar time to run and will carry out the agent’s commands.

To stop the simulation, hit `<ctrl-c>` in any agent window.

Clicking on the `Run` button is equivalent to typing `schedule`, and clicking on the `Stop` button is equivalent to hitting `<ctrl-c>`.
Basic operations, continued

After running for a while, stop SoarSF and click on the Goal Stack button. This will show you the goal stack of the agent in that window only (all of the other buttons effect the whole system). Compare the goal stacks of luke-1 and luke-2. Luke-1 is the lead and luke-2 the wingman. Notice that luke-2 is following the leader, whereas luke-1's goal stack is more complicated.

Finally, make the simulation run faster than real time by sliding the Speed Control slider to somewhere between 3.0 and 4.0. Because this increases the apparent time between ticks of the simulator, problems can arise when running too fast. Wingmen can lose their leaders and aircraft can collide. This is best used for flying long, straight flight paths and on lightly-loaded machines.

Changing the mission

Remove this scenario with New Scenario... and reload scenario.dat. In the window of luke-1, type mission voice clare, and in the window of luke-2 type mission voice randy. This will cause these agents to play sound files whenever they send a radio message. Luke-1 will use sounds recorded from Clare, and luke-2 will use sounds recorded from Randy.

Run this scenario and listen for the voices. It is more difficult to hear on some workstations due to the location of the speakers.
1.5: Using the Exercise Editor

Karen J. Coulter

3:15 - 4:30 April 29, 1996

Soar/IFOR Tutorial

University of Michigan AI Lab

Background

1. The ModSAF editor for generating scenarios is vehicle/unit based (bottom-up). Much of the exercise specification is the same for all vehicles; it's only at the Mission and Element levels that much changes. This organization of the ModSAF editor forces users to enter the same high-level data over and over again for each agent.

2. The Exercise Editor is organized in a hierarchical fashion, following the briefing structure used by the Navy (top-down). Its features include:
   - inheritance - lower levels inherit data values entered at higher levels
   - data driven - specifying a value for certain parameters can change how other parameters are displayed for input.
   - error checking - a limited amount of error checking has been implemented, alerting the user to inconsistencies in the data which could cause problems at execution time.
**Terminology**

**Exercise** - a series of planned engagements located in a specific geographical region. Typically lasts several days.

**Event** - all activities taking place during the same time period of an exercise. An Event typically lasts several hours. Usually many Events per Exercise. Identified by Event number.

**Mission** - activity organized toward achieving a specific objective, such as bombing a target. Each Mission has a designated Mission type and is identified by Mission number. Usually carried out by 1, 2 or 4 aircraft.

**Element** - the aircraft that carry out the Mission. Either a vehicle, a section (2 aircraft) or a division (4 aircraft).

The Exercise Editor has four levels of screens, corresponding to each of the objects above.

**Scenario** - ModSAF name for an Event. A scenario.dat file must exist for each event that is to be simulated in SoarSF.

---

**Exercise Editor Widget Set**

The Exercise Editor presents five basic widgets to the user for data entry:

- **Buttons**: Used to navigate from one level to another and to pop up/down related screens. Click with left mouse button to activate.

- **OptionMenus**: Displays the current setting for a parameter. Options which are not currently selected are only displayed when the menu is activated. To change a value, move mouse over menu, hold down left mouse button and release button over desired value.

- **PulldownMenus**: Used for various functions as indicated by the label on each PulldownMenu. To activate, move mouse over menu, hold down left mouse button and release over desired action.

- **TextEntry**: Used to enter free-form text values. To change, move mouse over text widget and click left button (highlight box will appear), and enter text from keyboard.

- **RadioButtons**: Displays the current setting for a parameter, but also displays all values not currently selected. To change value, click left mouse button over icon next to desired value.

There are also **FileSelectionBoxes** for reading/writing the various data files used by the Exercise Editor and SoarSF.
Exercise Level

The Exercise Level is the first screen you will see at startup.

Activating "Radio Color-Freq Chart" and "Waypoints..." buttons will pop up related displays for additional data entry. Use the "Edit Events" PulldownMenu to create a new Event or edit an existing Event.

Event Level

To move back to the parent Exercise Level, activate the button labelled with the exercise name, immediately following "Event Level."

Activating the "Code Words..." button will pop up a related display for additional data entry. The "Post Event Summary" button will pop up a display of Event information. Activating the "Save Scenario File" button will pop up a FileSelection Box for saving scenario and agent data. Use the "Edit Missions" PulldownMenu to create a new Mission or edit an existing Mission.
Mission Level

To move back to the parent Event Level, activate the button labelled with the event number, immediately following "Mission Level."

Activating the "Commit Criteria..." and "CAP Parameters..." buttons will pop up related displays for additional data entry. Use the "Edit Mission Elements" pulldown menu to create a new Element or edit an existing Element.

Element Level

To move back to the parent Mission Level, activate the button labelled with the mission number, immediately following "Element Level."

The vehicle CallSign is of particular importance: it will determine the name of the Soar agent and the directory name for the agent data file.

This is the lowest level screen in the Exercise Editor.
Briefing Status Tree Display for Navigation

The Briefing Status Tree displays all objects currently defined at each level for the current exercise. It is a separate screen that can be popped up at any time by activating the button labeled "View Briefing Tree" at the left side of the Editor window.

[Diagram of Briefing Status Tree]

Moving the mouse over the desired button and clicking the left mouse button will cause that particular object to be displayed in the main Editor window. Activating the "Rescan" button will update the information in the tree. Activating "OK" will pop down the screen.

Files Used by the Exercise Editor

There are four file types used by the Exercise Editor:

1. **exercise.dat** - Contains all data for the entire exercise; used only by the Exercise Editor. This file can be saved at any time by activating the "Save Exercise..." button at the left of the Exercise Editor screens. You will also be prompted to save this file whenever you quit the Exercise Editor.

2. **scenario.dat** - SoarSF scenario file. This file is only written out by the Editor, it is never read back in. Only SoarSF reads these files. It is saved from the Event Level, since events correspond to SoarSF scenarios.

3. **mission** files - SoarSF agent parameter files, which contain all data required by a particular agent to carry out its mission. This file is only written out by the Editor, it is never read back in. Only SoarSF reads these files. These files (one per agent) are generated whenever you save the **scenario.dat** file.

4. **terrain** files are read by both the Exercise Editor and SoarSF. They are part of the ModSAF distribution and are used to describe the terrain database for the exercise. You won't ever directly access these files from the editor, but both the editor and SoarSF are using terrain files whenever they run.

*Never change any of these files directly, always change them using the Exercise Editor.*
Directory Structure for the Exercise Editor

The Exercise Editor expects to be run from the same location where SoarSF is run. For this tutorial, it is run from the `~ifr/modsaf` directory.

In that directory, you will find the `ExerEdit` script to run the ExerciseEditor, and a subdirectory, `exercises/`, where all files generated by the Editor reside.

Each exercise has its own subdirectory, eg: `tutorial`, and in that subdirectory are the `exercise.dat` file (which has all the data for the exercise), and subdirectories for each event in the exercise `Event1`, `Event2`....

The Event subdirectories each have a `scenarios/` subdirectory with the file `scenario.dat`, (the file loaded by SoarSF), and subdirectories for each agent participating in the event, `agent-1/` `agent-2/` ...

Each agent subdirectory has two files in it: `.init.mission`, a startup file for loading agent productions into SoarSF; and `.mission`, the mission-specific data for the agent.

continued...

Directory Structure for the Exercise Editor, cont.

The following structure is what would exist for an exercise called `tutorial` that has one Event with two agents in that event: `dstar-1` and `luke-1`:

```
~ifr/modsaf/
  ExerEdit
  exercises/
    tutorial/
      exercise.dat
      Event1/
        dstar-1/
          .init.soar
          .mission
        luke-1/
          .init.soar
          .mission
    scenarios/
      scenario.dat
  ../../../terrain/
    a2-ocean.c4b
```

*NOTE:* This example is not reprinted accurately the format here is to make clear the organization of the files.
Running the EE: Editing an Existing Exercise

The Exercise Editor is invoked using the ExerEdit script:

```
ExerEdit -exercise {exercise-name} -event {event-number} [other args]
```

-exercise can be abbreviated as -ex, and -event can be abbreviated as -ev. The [other args] will be discussed at the end of this presentation.

To invoke the Editor for the tutorial exercise as described in the previous two slides, you would enter:

```
ExerEdit -ex tutorial -ev 1
```

At this point, the Editor will print some informational messages to the window where you invoked the script, and will start to load the exercise.dat file from the tutorial subdirectory. When the data file has been loaded, the Editor is ready. You may see only an outline of a window on the screen— if so, move the mouse to the upper left portion of your monitor and click the left button to anchor the Exercise Editor window.

continued...

Running the EE: Editing an Existing Exercise, cont.

You can now edit parameters and navigate the various levels as desired to modify the data.

When you have finished modifying parameters and the exercise is completely defined, you must save the scenario information and agent parameter files which are required by SoarSF. To do this, simply go to the Event level and activate the 'Save Scenario File' button. This will cause a FileSelectionBox to pop up. It will default to the correct location and filename for saving the required files. By activating the 'OK' button, both the scenario.dat file and the agent .mission files will be saved.

Individual agents/vehicles at the Element Level MUST have a CallSign entered in order for those agents to be included in the simulation. Agents with no CallSign will not have their data stored in the scenario.dat file, nor will their .mission file be generated.

To quit the Exercise Editor, move the mouse over the 'File' label in the top MenuBar, press the left mouse button, and release it over the Quit option. You will be prompted as to whether or not you wish to save the exercise.dat file. If you have made any changes to the data since the last time you saved this file, you will need to save it again in order to not lose those changes. The editor will not be able to tell you if the current file is up to date.
Running the EE: Creating a New Exercise

Creating a new exercise with the Exercise Editor is not much different from editing an existing exercise – it simply requires that these steps be performed before invoking ExerEdit:

1. Make sure you are in the same directory location as the ExerEdit script.
2. Choose a short, descriptive name for the exercise: \$name
3. Create the new exercise subdirectory: \$name
4. Determine which terrain file you wish to use for the exercise. Choose only from the files found in the ..../terrain directory.
5. Edit the ExerEdit script and add \$name as an option to the switch statement where terrain files are assigned and assign your selected terrain. (Do not include the filename extension of the terrain file.) This is most easily done if you can find an existing exercise which uses the same terrain file – just add \$name: to the line above that existing exercise name and the same terrain file will be used.
6. Edit the soarsf script in the same fashion, adding the same exercise name and assigning the same terrain file as you did in ExerEdit.

continued...

Running the EE: Creating a New Exercise, cont

You can now invoke the Exercise Editor for your new exercise:

    ExerEdit -ex \$name -ev 1

Even though no Event 1 yet exists, you can still specify it as the argument to ExerEdit.

Since there is no exercise.dat file to be loaded, the Exercise Editor will simply come up with only the top Exercise Level created. You must create the Events, Missions, and Elements required for your exercise. As each object is created, there will be default values for the parameters, which can either be left as they are (if they are valid for your exercise) or modified as needed. Be sure that each agent/vehicle you create has a CallSign at the Element Level.
Commandline Arguments to the Exercise Editor

The complete usage statement for the Exercise Editor is:

ExerEdit -help

-exercise can be abbreviated to -ex
-event can be abbreviated to -ev

-autoload off prevents the exercise.tcl file from being autoloaded
-real forces the use of a real terrain database, with all terrain structures.
-soar6 forces the Editor to write out Soar6-style agent files.
-help prints out this information about the Editor arguments.
2.1: Domain Description, Part Two
Groups of Aircraft

John E. Laird

April 30, 1996, 9:00am-9:45am

Soar/IFOR Tutorial
University of Michigan AI Lab

Command Groups

- Planes can be hierarchically organized into:
  individual vehicles, sections (2-ship), divisions (3/4-ships),
  packages (groups of sections and divisions).
- We also treat tanker plus division during refueling as a group.
- Each group has a lead, who is in control of the mission.
- In SoarSF, we represent:
  - All possible groups an aircraft could participate in (group).
  - All groups that are currently active (~active *yes*).
  - The primary group for tactics and maneuvering (group.primary).
Section/2-Ship Command Groups

- A section (2-ship) consists of a lead and a wingman.
- The lead is responsible for flying the section to the correct places, plans tactics for attacks, etc.
- The wingman is responsible for staying in formation. The lead will adjust only if necessary to help wingman. (Perform an S-turn if too far in front.)
- The lead is determined before take-off, and is unlikely to change except for damage or absence of weapons.

Section/2-Ship Formations

- combat spread: defensive and offensive (also called line abreast)
- fighting wing, bearing, cruise, wedge, trail, parade.
Section Flying in Formation

There is a "box" that the wing tries to fly in defined by lateral range, lateral separation, vertical separation, and target aspect.
1. If very far away, just fly pursuit (toward the lead).
2. If out of lateral separation, turn a bit.
3. If out of target-aspect, adjust speed.
4. If out of vertical separation, adjust altitude.
   If low altitude, fly above or co-altitude.

Section Formations: Tactical Turns
Section Engagements

- Lead will decide to commit (using defensive combat spread).
- Will switch to offensive combat spread when get close.
- Lead will "sort" targets.
- Wingman will stay with lead, but try for shots of opportunity.
- Wingman will "beam" with lead if necessary.

Section Engagements: Pincer

- Lead with decide to commit.
- Lead and wing will split at given range to bogeys.
Division Formations

- Division Lead (1) is flying mission flight plan. primary group is the division.
- First wingman (2) is flying off of division lead (1). primary group is its section.
- Second lead (3) is flying off division lead (1). primary group is the division.
- Second wingman (4) is flying off second lead (3) primary group is its section.

Division Formations

- Can dynamically change formations.
- If one plane is lost, change to VIC formation.
- If lead is lost, reorganize with new lead (3).
- Everyone tracks progress of mission through waypoints.
- Section leads may strip at contact point for separate attacks.
- Division will rejoin following an attack.
Package Formations

- Consists of multiple sections and/or divisions.
- Each division/section of the formation has its own lead.
  - These leads have the package as their primary group.
  - These leads may strip from package to attack separate targets or to intercept bandits.
Forming up Groups

- Sections form up as soon as possible following take-off: often launch at same time.

- Divisions form up through running rendezvous (flying to a point)
or by circling at a specified altitude at a point (often the carrier).
  In Air Force, will take-off together.

- Packages will often meet at a refueling, holding point.
  Lead will call “strikers commencing” when time to leave, and package will form up based on his position.
2.2: Domain Description, Part Four
Groups of Aircraft Interacting with Controllers

Paul E. Nielsen

April 30, 1996, 9:45-10:15am

Soar/IFOR Tutorial
University of Michigan AI Lab

Overview

This talk addresses the current roles of control agents and their interactions with Navy fixed wing aircraft.

Soar/IFOR control agents are designed to support all control functions and interactions with the flying agents.

These control agents are instantiated differently than flying agents, but use the same generic underlying code.

All communication is done in an understandable language over simulated radios.
Types of Control Agents

Separate control agents primarily interact with the flying agents for two classes of mission:

- Defensive counter-air (DCA)
  - Air intercept controller (E-2C or GCI)
- Close air support (CAS)
  - Mission requestor
  - Agents to deconflict flight through combat intensive areas
  - Guide flight to target

Soar-IFOR Control Agents

- Defensive counter-air
  - Air intercept controller
    * AEW (E-2C)
    * GCI
- Close air support
  - Mission initiation and guidance to target
    * FAC/FAC(A): Final control
  - Deconfliction
    * TACC: Primary control
    * TAD: Enroute control
    * DASC: Procedural control
    * FSCC: Fire support control
Sample Scenario

--- original flight path
--- actual flight path

Control Agents’ Capabilities

- Superior information gathering through
  - Proximity
  - Superior sensors

- Limited weapons
  - Typically no weaponry of their own
  - Every fighter in the sky is its weapon
The Roles of Control Agents

There are eight basic categories of responsibilities for control agents.

1. Maintain control of own vehicle
2. Controlling missions
3. Handling threats
4. Deconfliction
5. Planning
6. Resource allocation
7. Interaction with other control agents
8. Damage assessment

Maintain control of own vehicle

In addition to their control functions Soar/IFOR control agents must navigate their own vehicles.

- Ship based controllers and the GCI are easy because they do not currently move.
- The E-2C flies a racetrack pattern and will take evasive action if threatened.
- The FAC(A) flies a complex mission which actively searches for targets while avoiding becoming a target of enemy or friendly fire.
Controlling Missions

Soar/IFOR control agents work with the pilots of FWA to direct their missions.

Because all FWA pilots are officers, they do not need to be micro-managed as much as a tank platoon might.

There are three aspects to controlling a FWA mission:

- Mission initiation: Calling a plane from its post to attack a target
- Mission modification: Changing route, target, and station information
- Monitoring mission progress: Confirming each plane is where it is supposed to be.

Handling Threats

Because of their superior information gathering capabilities, control agents are typically aware of potential threats long before fighter or attack aircraft.

The Soar/IFOR control agents are responsible to the fighter and attack aircraft for the following information about threats.

- Threat detection and acquisition
- Threat tracking: Provide continuous updates about target movement
- Threat identification (friend, foe, or other): Depending on ROE, pilots may not be authorized to prosecute a target unless cleared by a control agent.
Deconfliction

Kills due to friendly fire are qualitatively worse than enemy kills. In addition to losing the aircraft they destroy moral.

Soar/IFOR controllers minimize the chance of friendly kills through the following:

- Altitude deconfliction: Simple air traffic control by assigning each mission a unique altitude.
- Deconflicting routes: Maintain a clockwise or counterclockwise flow to the airspace so that friendly planes do not interfere with each other.
- Synchronization: Use precise timing information to avoid friendly fire.

Planning

Soar/IFOR control agents do not currently perform the detailed mission planning of a human controller, but provide a simple framework which may be extended.

- Mission planning: First come, first served mission allocation.
- Route planning: Select best from preplanned set of points for target approach and egress.
- Planning own actions
Resource Allocation

The Soar/IFOR agents may assign themselves to missions or rely on a control agent to task them.

- Decentralized allocation
  - For air intercept planes may have commit criteria
  - Free to engage when criteria are met without specifically being told
- Centralized allocation
  - Always used for CAS because of integration necessary
  - May be used for air intercept with controller calling bogey assignments

Interaction with Other Control Agents

In addition to directing fighter and attack missions, control agents must coordinate their actions with other controllers.

- Relaying requests
  - Someone on the front lines requests fire support from FSCC
  - FSCC determines air and requests planes from DASC
  - DASC either allocates aircraft or requests launch
- Advising on aircraft availability
  - Once mission is allocated, information about mission flows back to requestor and intermediate controllers.
- Informational
  - Reconnaissance information gets shared between controllers
Damage assessment

Battle damage assessment is a special kind of informational reconnaissance data which is shared among the controllers.

It is relayed by the plane making the attack rather than between the controllers.

Both the plane and the controller on the scene assess the extent of the damage for this report.

Control Structures

Control agents maintain specialized knowledge structures for other agents and groups of agents:

- Command groups
  - Groups of agents flying same mission
- Control groups
  - Groups of agents expected to interact with this controller
- Current mission
  - Transient mission under control for brief period
- Other controllers
  - Mission roles
  - Radio frequencies
  - Existence
Message Generation

Communication by understandable language

Pilot / controller communication is highly stylized and restricted compared to casual conversations.

- Templates are taken from actual pilot/controller communication
- Actual values are replaced with variables
- Few special cases require more sophisticated processing

Communication

Your bogey is at bearing <bearing> range <range> miles angels <altitude> heading <heading>
Summary

Soar/IFOR are autonomous entities, functioning without human intervention, which can direct Navy (and soon Airforce) fixed wing aircraft toward successful resolution of their assigned missions.

We have currently implemented a number of Navy and Marine controllers which provide the infrastructure necessary for successful intercept and attack missions.
Exercise 2.1: Adding New Mission

John E. Laird

April 30, 1996, 11:00am-12:00pm

Soar/IFOR Tutorial

University of Michigan AI Lab

Create a New Event: Sweep

You now need to create a mission to attack the carrier.

1. Create a new exercise, call it attack-<group-letter>, event 1 with start time 100.

2. This will be a strategic attack mission with MiG-29’s from home-base to alexis, fredrick, kiev, the target, and then moscow then homebase:
   home-base 36 00 00 N, 116 55 00 W
   alexis (contact point) 35 05 00 N, 119 25 00 W
   kiev (initial point) 32 46 00 N, 118 50 37 W
   target 32 9 52 N, 118 50 1 W
   moscow 33 42 00 N, 117 44 00 W.

3. Set the target type as carrier.

4. Load the scenario, change the MiG color, resave the scenario.

5. Run this over the net with the other scenario you generated above. For the command line of running soarsf, use
   -exercise attack-<group-letter> -event 1 -group <number> -network -database <letter>
   Use the same <number> for all machines running in your scenario (but different from everyone else’s).
   Make sure you use a different letter with -database for each machine.
2.3: Overview of Problem Spaces in TacAir-Soar

John E. Laird

April 30, 1996, 11:30-12:00pm

Soar/IFOR Tutorial

University of Michigan AI Lab

Organizing Behavior

• How do we organize behavior?

• As a series of decisions about what to do next. For people, this seems to happen every 50msec. We abstract up to every 200-500msec.

• For planning: what am I going to do in the future?

• For action: what should I do right now.

• We call each action an operator. Sometimes these become goals when they cannot be performed right now.

• We organize operators into problem spaces: operators that can apply together to achieve a goal.
Example Problem Space I: Fly-Flight-Plan

Consider the goal of the lead of a section flying the flight plan of a mission. What actions can be performed?

- Fly the current leg of the route (fly-control-route).
- Switch to next leg of route.
- Do everything that has to be done at a waypoint (fly-control-point).
- Switch to next waypoint of route.
- Accept changes to the mission from a controller (fly-control-route).
- Following an intercept, reconsider which is next waypoint (intercepted-route).
- ...

Example Problem Space II: Fly-control-route

Fly a route using appropriate altitude, speed, tactics, etc.

- Set up the waypoint computer so it tells us which way to fly (waypoint-computer)
- Control the aircraft so it flies in the right direction, at the right speed, and the right altitude (fly-tactic)
- For an attack, fly the appropriate attack tactics in terms of geometry, entry, and delivery (fly-attack).
- Communicate with controllers along the route (communicate-contact).

fly-tactic used in many different problem spaces.
Operators for all Goals

These operators should be available at any time.

- Authentication
- Modify command structure (activate a section/division, change-lead, ...).
- Change formation.
- Communicate with partner about position – check in with partner.
- Adjust radar.
- Detect low on fuel.
Summary

• There is a hierarchy of “operators” with some acting as goals to help control the selection of primitive operators.

• The selection of an operator is separate for performing the action. There can be many different reasons for doing an action and an action can be done differently in different situations.

• Most operators are tied to their goals/problem spaces, but some of them “float” and can be used for any goal.
2.4: Introduction to Soar

Clare Bates Congdon
(with TacAir-Soar examples from Randy Jones)

April 30, 1996, 1:00-3:15

Starting to Learn Soar

Step 1: For now, forget about sequential programming languages; Soar is different:
  • No procedures, subprocedures, etc.
  • The execution of a Soar program is not sequential

Step 2: For now, think about problem solving instead of programming.
Problem Solving in Soar

All problem-solving activity in Soar is formulated as selecting and applying operators to a state.

Definitions:

*state* — a representation of a problem-solving situation
*operator* — makes changes to a state
*goal* — a desired situation
*problem-space* — a set of (virtual) states and operators for the task

Many operators may need to be applied to a state in order to achieve the goal, but operators are selected and applied one at a time.

(Problem-spaces are the most difficult of the above for novices to grasp, so give it some time to sink in.)

Blocks-World Example: States and Operators

States are representations of a problem-solving situation
Operators make changes to a state
Blocks-World Example: Goal

The goal in the blocks-world task is to stack the three blocks with A on the top and C on the bottom:

```
A  B  C
```

Initial State

```
A    C
```

Goal

The initial state is the first state description in the task.
The goal is sometimes called the desired state.

---

Blocks-World Example: Problem Spaces

The problem-space in the blocks-world includes all operators that move blocks from one location to another location and all possible configurations of the three blocks.

```
A B C
```

STATE = states

OPERATOR = operators

NOTE: Because this is such a simple problem, we can illustrate all possible states and all possible operators, but this is not always the case. In general, the set of possible states and the set of possible operators may be infinite.
Blocks-World Example: Problem Spaces, part 2

Examples of states and operators that do not appear in the blocks-world problem space:

**Illegal States**

- B C
  - two blocks

- Z A C
  - a block named Z

- D B C
  - four blocks

**Illegal Operators**

- move two blocks at once
- change the name of a block
- add a block
- remove a block

*NOTE:* To say that these states and operators do not exist in this problem space is not to say that we can't imagine states and operators or that they couldn't exist in a different problem space.

---

Problem Spaces in TacAir-Soar

**Execute-Mission Problem Space**

**Operators**

- Change-Mission
- Fly-Flight-Plan
- Follow-Leader
- Intercept
- Racetrack
- Return-To-Base

---

Soar/IFOR Tutorial

2.4: Introduction to Soar
Soar’s Decision Cycle

As Soar runs, it executes a series of decision cycles. During each decision cycle, Soar tries to select a new operator. The selection of a new operator is called a decision.

In Soar, the selection and application of operators are two distinct activities. An operator gets applied at the beginning of the decision cycle that follows its selection (during the "chugging").

NOTE: Soar always tries to select an operator, but is not always able to do so. We’ll discuss this later.
The operators are selected at random, so the trace you see when you actually run Soar is likely to differ from what’s shown here.

Hands-On: Run the Blocks World

The output from Soar is often called a “trace”, so this can be called a trace of the blocks-world task. The execution of a Soar program consists of a number of decision cycles; in the blocks-world task above, Soar completes four decision cycles (the output from each decision cycle is preceded by a number):

0. The “zeroth decision cycle” isn’t really a decision cycle. This is when Soar creates the first state (with minimal structure).

1. In the first decision cycle, Soar (randomly) selected an operator to move block A to ontop of block C.

2. In the second decision cycle, Soar (randomly) selected an operator to move block A to the table.

3. In the third decision cycle, Soar (randomly) selected an operator to move block B to ontop of block C.

4. In the fourth decision cycle, Soar (randomly) selected an operator to move block A onto block B.

The Soar program recognizes that this is the tower it was trying to build; it has achieved the goal of building the A-B-C tower. Therefore the program tells Soar to halt — to stop running the Soar program.
Working Memory

Operators and states are represented in *Working Memory*, which represents the current problem-solving situation.

For example, in the initial state in the blocks world, working memory contains information such as:

There are three blocks and a table in the state. The blocks are named A, B, and C. The blocks are all on the table. The blocks are all clear. (They have nothing on top of them).

The structure of the initial state is created by the Soar program; no operators can be suggested until the initial state has been created.

*NOTE:* Creating the initial state in working memory is similar to initializing variables in a sequential programming language.

Operators in Working Memory

Once the initial state is created in working memory, the Soar program can suggest possible operators. There will be six possible operators for the initial state:

Maybe move block A ontop of block B.
Maybe move block A ontop of block C.
Maybe move block B ontop of block A.
Maybe move block B ontop of block C.
Maybe move block C ontop of block A.
Maybe move block C ontop of block B.

All of the suggestions are added to working memory as *proposed* operators, but only one will be chosen (during the decision cycle) as *the* operator.
Operators in Working Memory

If multiple operators are proposed, the Soar program should suggest some means of comparing these operators.

For this simple task, the program just says that all operators are the same (indifferent), which will allow Soar to pick one at random.

Moving block A ontop of block B is as good as anything else.
Moving block A ontop of block C is as good as anything else.
Moving block B ontop of block A is as good as anything else.
Moving block B ontop of block C is as good as anything else.
Moving block C ontop of block A is as good as anything else.
Moving block C ontop of block B is as good as anything else.

Back to the Decision Cycle

After one of the proposed operators is selected (maybe at random), we’ve completed the first decision cycle:

*Blocks–World*

**Decision Cycle**

#1

- propose operators
- create the initial state
- make operators indifferent
- select the first operator

We have not yet *applied* any operators. That won’t happen until the second decision cycle.... And we won’t talk about that just yet.
Working Memory in TacAir-Soar

**Active Goals**
- Execute-Mission
- Intercept
- Employ-Weapons
- etc...

**Data Structures**
- **Static Parameters**
  - Vehicles
  - Weapons
  - Sensors
  - etc....
- **Dynamic situation descriptions**
  - Agents
  - Geometries
  - Threats
  - Weapons
  - etc....

**Perception**
- Radar Inputs
- Vision Inputs
- Radio Messages

**Action**
- Press a button
- Fly the aircraft

What is not in working memory:
- Doctrine and Tactics: How to fly missions.

---

Working Memory Elements

Working memory holds *working memory elements* (WME’s):

- WME’s have identifiers, attributes, and values.
- Groups of WME’s with the same identifier form *objects* in working memory.
- An identifier is nothing more than a label that connects different statements about the same object.

### An Abstract View of an Object in Working Memory

<table>
<thead>
<tr>
<th>identifier</th>
<th>attribute</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>isa</td>
<td>state</td>
</tr>
<tr>
<td>S1</td>
<td>name</td>
<td>top-state</td>
</tr>
<tr>
<td>S1</td>
<td>operator</td>
<td>O2</td>
</tr>
<tr>
<td>S1</td>
<td>thing</td>
<td>B1</td>
</tr>
<tr>
<td>S1</td>
<td>thing</td>
<td>B2</td>
</tr>
<tr>
<td>S1</td>
<td>thing</td>
<td>B3</td>
</tr>
<tr>
<td>S1</td>
<td>thing</td>
<td>T1</td>
</tr>
</tbody>
</table>

*These values are identifiers of other objects in working memory, and indicate links in working memory.*
Details of Working Memory

- Elements in working memory are linked.
- Links are one-way connections that represent substructure.
- Everything in working memory is linked to a state (perhaps indirectly).
- The current operator is substructure of the state — that is, it is linked to the state.

NOTE: Operators that have been proposed but not yet selected are also substructure of the state. But this distinction gets confusing, so we'll try to put it aside for now.

Production Memory

Knowledge of how to select and apply operators is represented in Production Memory:

- Production memory represents the Soar agent’s long-term knowledge.
- Each production has conditions and actions.
- Productions are not connected or related to one another.

NOTE: Productions are what you write when you write a Soar program.
Productions

Production memory contains productions:

- Productions are similar to if-then rules:
  
  \textit{if} the conditions match working memory
  \textit{then} the action is executed.

- The conditions of productions are compared to working memory.

- Most actions make suggestions for changes to working memory. (But do not change working memory directly.)

\textbf{Example production in English:}

\begin{itemize}
  \item If there is a state and the state has a block and the state has a block and the two blocks are different and the first block is clear and the second block is clear
  \item Then suggest an operator to move the first block on top of the second block
\end{itemize}

\textbf{Example TacAir-Soar Production}

If I am executing a mission that may involve an intercept and I am not already intercepting a bogey group and there’s a bogey group with a member that has satisfied my commit criteria and I am the lead of my commit group

Then suggest an operator to select the group to intercept
Variables in Productions

Productions use *variables*:
- Variables match against identifiers and constants in working memory.
- Variable bindings must be consistent within a production.
- Variable bindings within a production need not be unique.

Example production in pseudo-code

```
<state1> is a state
<state1> has a thing <block1>
<state1> has a thing <block2>
<block1> ≠ <block2>
<block1> is clear
<block2> is clear

suggest an operator to move <block1> on top of <block2>
```

Production Instantiations

A production is said to be *instantiated* with a specific set of variable bindings/WME's:
- When the production is *matched* with a consistent set of bindings, that instantiation will *fire*.
- A production may fire only once for a given instantiation.
- All new instantiations fire in parallel (their actions are executed).
- Some instantiations retract their actions when they no longer match.

*NOTE:* We'll discuss retraction later.

*NOTE:* We sometimes get sloppy in our language and say "production" when we mean "production instantiation". Call us on it.
Illustration of Multiple Instantiations

Example production

- state 1 is a state
- state 2 has a thing *clock1*
- *clock2* is a clear

Each production may have multiple instantiations

(All instantiations of a production will fire, so this single production suggests several operators)

Example instantiation

S1 is a state
S1 has a thing B1
S1 has a thing B2
B1 ≠ B2
B1 is clear
B2 is clear

suggest an operator to move B1 on top of B2

Example instantiation

S1 is a state
S1 has a thing B1
B1 ≠ B2
B1 is clear
B2 is clear

suggest an operator to move B1 on top of B2

Example instantiation

S1 is a state
S1 has a thing B1
S1 has a thing B2
B1 ≠ B3
B2 is clear
B3 is clear

suggest an operator to move B1 on top of B3

Example instantiation

S1 is a state
S1 has a thing B1
S1 has a thing B2
B1 ≠ B3
B2 is clear
B3 is clear

suggest an operator to move B2 on top of B3

Example instantiation

S1 is a state
S1 has a thing B1
S1 has a thing B2
B1 ≠ B3
B2 is clear
B3 is clear

suggest an operator to move B1 on top of B2

2.4: Introduction to Soar

The example production matches the initial state in working memory and proposes six move-block operators.

These are six different instantiations of the production, but these six instantiations all *match* at the same time and all *fire* at the same time.

That is, the six operator proposals all happen at the same time in the Soar decision cycle (during the same “chug”).

Hands-On: Run the Blocks World Again

To run the blocks-world task again, we could quit Soar, start it again and load the blocks world (as on slide 10).

Instead, we'll reinitialize the blocks-world task, by using *init-soar*. This empties working memory and preference memory.

**soar> init-soar**

Then we can set the *watch* level to see more detail about what is happening when Soar runs. This time, we'd like to watch the productions that are firing:

**soar> watch productions -on**

Then we'll run Soar for one decision cycle (the trace is shown on the next slide):

**soar> run 1 d**
Hands-On: Run the Blocks World Again

0: => S: S1
Firing blocks-world*elaborate*initial-state <-- printed by Soar

Initial state has a, b, and c on the table. <-- printed by
Firing blocks-world*propose*move-block a production
Firing blocks-world*propose*move-block
Firing blocks-world*propose*move-block
Firing blocks-world*propose*move-block
Firing blocks-world*propose*move-block
Firing blocks-world*propose*move-block
Firing blocks-world*propose*move-block
Firing blocks-world*propose*move-block
Firing blocks-world*propose*move-block
Firing blocks-world*select*move-block*indifferent
Firing blocks-world*select*move-block*indifferent
Firing blocks-world*select*move-block*indifferent
Firing blocks-world*select*move-block*indifferent
Firing blocks-world*select*move-block*indifferent
Firing blocks-world*select*move-block*indifferent
Firing blocks-world*select*move-block*indifferent
Firing blocks-world*select*move-block*indifferent

1: 0: 05 (move-block)

PREference Memory

The actions of productions create suggestions for changes to working memory; these are represented in Preference Memory.

- There is at least one preference for every WME.
- **Except:** Soar I/O can change working memory directly.

*NOTE:* We'll discuss I/O later.
Preferences

Preference memory contains preferences:

- preferences, identifiers, attributes, and values
- Each preference corresponds to a potential WME — if the preference says that the WME is acceptable and there are no conflicting preferences, the WME will be added to working memory.
- Conflicting preferences are resolved by the Soar "decision procedure”.

An Abstract View of Preferences:

+ S1 is a state
+ S1 is named top-state
+ S1 has an operator O2
  + S1 has a thing B1
  + S1 has a thing B2
  + S1 has a thing B3
  + S1 has a thing T1
  & S1 has a thing B1
  & S1 has a thing B2
  & S1 has a thing B3
  & S1 has a thing T1

Grouping preferences into objects (by identifier only) is not often useful — it's more useful to group preferences by identifier and attribute. (If there are any conflicts in preference memory, they will appear within the identifier-attribute group.)

Preferences for TacAir-Soar Operators

All operators in TacAir-Soar have preferences that say they are acceptable and indifferent.

Some operators have additional preferences, depending on the priority of the operator:

High-priority operators: also have best preferences.

push-fire-button, change-formation,
notice-bogey-fired-missile

Medium-priority operators: have no additional preferences.

intercept, employ-weapons, get-missile-lar

Low-priority operators: also have worst preferences.

identify-contact, forget-bogey
Example Preference Resolution for TAS Operators

<table>
<thead>
<tr>
<th>Proposed operators</th>
<th>Priority</th>
<th>Preferences</th>
<th>Winner</th>
</tr>
</thead>
<tbody>
<tr>
<td>push-fire-button</td>
<td>high</td>
<td>best, indifferent</td>
<td>← one of the three best</td>
</tr>
<tr>
<td>intercept</td>
<td>medium</td>
<td>indifferent</td>
<td>operators will be chosen at random</td>
</tr>
<tr>
<td>employ-weapons</td>
<td>medium</td>
<td>indifferent</td>
<td></td>
</tr>
<tr>
<td>identify-contact</td>
<td>low</td>
<td>worst, indifferent</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Proposed operators</th>
<th>Priority</th>
<th>Preferences</th>
<th>Winner</th>
</tr>
</thead>
<tbody>
<tr>
<td>push-fire-button</td>
<td>high</td>
<td>best, indifferent</td>
<td>← one of these three</td>
</tr>
<tr>
<td>change-formation</td>
<td>high</td>
<td>best, indifferent</td>
<td>operators will be chosen at random</td>
</tr>
<tr>
<td>notice-bogey-f-m</td>
<td>high</td>
<td>best, indifferent</td>
<td></td>
</tr>
<tr>
<td>employ-weapons</td>
<td>medium</td>
<td>indifferent</td>
<td></td>
</tr>
<tr>
<td>identify-contact</td>
<td>medium</td>
<td>indifferent</td>
<td>← one of these two</td>
</tr>
<tr>
<td>forget-bogey</td>
<td>low</td>
<td>worst, indifferent</td>
<td>operators will be chosen at random</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Proposed operators</th>
<th>Priority</th>
<th>Preferences</th>
<th>Winner</th>
</tr>
</thead>
<tbody>
<tr>
<td>intercept</td>
<td>medium</td>
<td>indifferent</td>
<td>← one of the three best</td>
</tr>
<tr>
<td>employ-weapons</td>
<td>medium</td>
<td>indifferent</td>
<td>operators will be chosen at random</td>
</tr>
<tr>
<td>get-missile-lar</td>
<td>medium</td>
<td>indifferent</td>
<td></td>
</tr>
<tr>
<td>forget-bogey</td>
<td>low</td>
<td>worst, indifferent</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Proposed operators</th>
<th>Priority</th>
<th>Preferences</th>
<th>Winner</th>
</tr>
</thead>
<tbody>
<tr>
<td>identify-contact</td>
<td>low</td>
<td>worst, indifferent</td>
<td>← one of the three best</td>
</tr>
<tr>
<td>forget-bogey</td>
<td>low</td>
<td>worst, indifferent</td>
<td>operators will be chosen at random</td>
</tr>
</tbody>
</table>

Soar/IFOR Tutorial

2.4: Introduction to Soar

Representation of Goals and Problem Spaces

Goals and problem spaces might be explicitly represented in working memory (as substructure of the state), or might be implicit.

- A goal can be explicitly represented in working memory as substructure of the state or operator (called a "desired state"), and achievement of the goal is recognized by productions that compare the desired state to the current state.

- A goal could also be implicitly represented through the use of productions that test the state. In this case, the description of the goal is written into the productions.

All goals in TAS are implicit.

NOTE: For most tasks, there are many possible state descriptions that achieve the goal. That is, the goal corresponds to a partial state description that does not refer to all possible attributes of the state.

Soar/IFOR Tutorial

2.4: Introduction to Soar
Goals in TacAir-Soar

**Execute-mission** terminated if:
- mission is complete, *or*
- my vehicle is destroyed.

**Employ-weapons** terminated if:
- there is no more threat, *or*
- we have lost contact with the threat, *or*
- something more important needs to be done.

**Launch-missile** terminated if:
- there is a missile in flight, *or*
- weapons envelope is no longer achieved, *or*
- missile is radar-guided and radar contact is lost.

Representation of Problem Spaces

An implicit problem space is often used in simple tasks, such as the blocks world:
- In this case, there is only one problem space used in the task.
- The productions in such a task suggest only those states and operator descriptions that are in the problem space.

An explicit problem space is represented as substructure of the state:
- Syntactically, it is just a label.
- Functionally, it serves to restrict the operators that are suggested (because productions that suggest operators check for the appropriate problem space).

In TacAir-Soar, each problem space is implemented as a sort of high-level operator; once this high-level operator is selected, it relies on other operators to actually do the work.
An Abstract View of Soar's Memories

MATCH
working memory elements match against productions, causing them to fire (working memory elements that are no longer present lead productions to retract)

FIRE & RETRACT
productions fire, creating preferences (and productions retract, removing preferences)

DECIDE
preferences are resolved, which leads to the creation of new working memory elements (and the removal of old working memory elements)

The Soar Elaboration Cycle

MATCH: Productions are compared to working memory to find new matches and new retractions.

Fire/Retract: Productions fire, creating new preferences; some productions retract, removing preferences.

Decide: Preferences are reevaluated, possibly changing working memory.
Soar's Elaboration Cycle

The Elaboration Cycle repeats until there are no more instantiations ready to fire or retract.

This repetition is called the *Elaboration Phase* of the Decision Cycle, and corresponds to the "chug, chug, chug" in the illustration on Slide 9.

Operator preferences are not evaluated in the elaboration cycle. They are evaluated in the decision phase of the decision cycle.

---

Five Problem-Solving Functions in Soar

**Selecting an operator:**
1. Proposing candidate operators  
2. Comparing candidate operators  
3. Selecting a single operator

**Applying an operator:**
4. Applying the operator  
5. Terminating the operator

Four of these functions involve the firing of productions (P), and subsequent creation of preferences.

The third function involves making a decision (D), accomplished via the evaluation of preferences.
Example: Five Problem-Solving Functions in Soar

We’ve already seen operator proposal, comparison, and selection in the simple blocks-world.

```
<table>
<thead>
<tr>
<th>Decision Cycle #1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. create the initial state</td>
</tr>
<tr>
<td>2. propose operators</td>
</tr>
<tr>
<td>3. make operators indifferent</td>
</tr>
</tbody>
</table>

select the first operator

<table>
<thead>
<tr>
<th>Decision Cycle #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. apply the first operator</td>
</tr>
<tr>
<td>2. terminate/retract proposals</td>
</tr>
<tr>
<td>3. the first operator</td>
</tr>
<tr>
<td>4. some of the old operator proposals</td>
</tr>
</tbody>
</table>

select the second operator

These all happen at the same time
```

In the second decision cycle of the blocks-world,
1. Several productions fire to apply the operator (next slide).
2. The application of the operator changes the state, which leads to more productions to fire and retract at once:
   (a) The current operator is terminated.
   (b) Some of the operator proposals are retracted.
   (c) Some new operators are proposed.

Example Operator Application

In the blocks world, there are four productions involved in applying the move-block operator. Very roughly, these are:
1. If a block has been moved, it’s no longer where it used to be.
2. If a block has been moved, it’s now in it’s new place.
3. If a block has been moved, where it used to be is now empty.
4. If a block has been moved, where it is now is no longer clear.

When these productions fire, they create preferences such as:
1. The former ontop should be rejected.
2. A new ontop should be proposed.
3. A new clear should be proposed.
4. The former clear should be rejected.

Since the table is always considered “clear”, all four of these productions are not instantiated when a block is being moved to or from the table.
Example Operator Termination

- The production that terminates the `move-block` operator just compares the operator to the state.
- The substructure of the operator says that a specific block should be moved to a specific location; this is compared with the actual location of the block.

Operator-termination productions create a preference that tells Soar that the operator has completed. (Called a `reconsider` preference.)

Why do we have to explicitly terminate the operator?
- In the blocks-world, the operator is applied quickly. But in some tasks, it may take several rounds of production firings/retractions (chugs) before an operator is fully applied.
- Worse case: the operator doesn’t finish — you'd want to know this.

---

Problem-Solving Functions in the Decision Cycle

The problem-solving functions do not happen in numerical order in the decision cycle:
- Proposal, comparison, application, and termination happen during the elaboration phase, and selection happens during the decision phase.
- Application and termination of the current operator might happen *in parallel* with proposal and comparison for the next operator.
Hands-On: Blocks World by Elaboration Cycles

Now, we'll run the blocks-world task looking at the productions that fire during each elaboration cycle.

Reinitialize Soar and source the file `blocks-monitor.soar`. This is a copy of the Soar program we've been running, but it's changed so that each production prints out a line when it fires.

```
soar> init-soar
soar> source blocks-monitor.soar
#*#*#*#*#*#*#*
soar>
```

The `blocks-monitor.soar` program has the same number of productions as the `blocks.soar` program — and these productions use the same name. When you overwrite a production in Soar — *using exactly the same name* — the old production is removed (excised) and the new one takes its place.

Soar prints a hash mark `#` for every production excised, and an asterisk `*` for every production loaded.

*NOTE:* You could also remove all productions at once by typing "excise -all".

Blocks World by Elaboration Cycles, Decision Cycle 1

We also have to reset the watch level so that Soar is no longer printing production firings:

```
soar> watch 1
```

Now, we'll run Soar one elaboration cycle at a time to watch the productions that are firing and retracting:

```
soar> run 1 e

0: ==S: S1
Making ACCEPTABLE and PARALLEL preferences for contents of initial state.
(All blocks, ontops, and clears.)
Initial state has a, b, and c on the table.

soar>
```
Blocks World by Elaboration Cycles, Decision Cycle 1

soar> run 1 e
Making an ACCEPTABLE preference for an OPERATOR (move-block: b to a)
Making an ACCEPTABLE preference for an OPERATOR (move-block: c to a)
Making an ACCEPTABLE preference for an OPERATOR (move-block: a to b)
Making an ACCEPTABLE preference for an OPERATOR (move-block: c to b)
Making an ACCEPTABLE preference for an OPERATOR (move-block: a to c)
Making an ACCEPTABLE preference for an OPERATOR (move-block: b to c)

soar> run 1 e
Making an INDIFFERENT preference for an OPERATOR (move block: b to a)
Making an INDIFFERENT preference for an OPERATOR (move block: c to a)
Making an INDIFFERENT preference for an OPERATOR (move block: a to b)
Making an INDIFFERENT preference for an OPERATOR (move block: c to b)
Making an INDIFFERENT preference for an OPERATOR (move block: a to c)
Making an INDIFFERENT preference for an OPERATOR (move block: b to c)

soar> run 1 e
  1:  0: 02 (move-block a to c)

Blocks World by Elaboration Cycles, Decision Cycle 2

soar> e 1
Making an ACCEPTABLE and PARALLEL preference for
    ONTOP (top-block: a bottom-block: c)
Making a REJECT preference for ONTOP (top-block: a bottom-block: c)
Making a REJECT preference for CLEAR (block: c)

soar> e 1
Making a RECONSIDER preference an OPERATOR (move-block: a to c)
Making an ACCEPTABLE preference for an OPERATOR (move-block: a to table)
Making an ACCEPTABLE preference for an OPERATOR (move-block: a to b)

(Rejections of five operator proposals do not appear in this trace)

soar> e 1
Making an INDIFFERENT preference for an OPERATOR (move block: a to table)
Making an INDIFFERENT preference for an OPERATOR (move block: a to b)

soar> e 1
  2:  0: 08 (move-block a to table)
Discussion of Blocks-World by Elaboration Cycles

The First Decision Cycle

Initially, Soar automatically creates an initial state; this is not technically a “decision”, since it happens automatically and without deliberation, but it is sometimes called “the zeroth decision”. The initial state automatically created by Soar does not yet contain blocks.

In the first elaboration cycle, representations of all the blocks and the table are added to the state, as are the relationships, such as that block A is on the table. These additions to the state are called augmentations, and are represented as working memory elements.

In the second elaboration cycle, several different operators are proposed. All of these operators are named move-block, but each moves one of the three blocks to the top of one of the remaining two blocks. For example, there are two move-block operators created to move block A; one moves block A to ontop of block B and the other moves block A to ontop of block C. There is no operator proposed to move block A to the table because block A is already on the table, but in general, there might also be move-block operators that move blocks to the table. All of these operators are named move-block, but they move different blocks to different locations; these are called different instantiations of the move-block operator.

In the third elaboration cycle, each of the move-block operators proposed in the previous elaboration cycle are made indifferent.

The first decision cycle ends when there are no more productions ready to fire or retract. Soar (randomly) selects one of the six move-block operators. This is now the current operator.

Discussion of Blocks-World by Elaboration Cycles

The Second Decision Cycle

The second decision cycle is similar to the first, except for the beginning.

In the first elaboration cycle (of the second decision cycle), three productions fire to apply the move-block operator, and moves block A ontop of block C. The preferences created lead to changes in the state: block A is no longer ontop of the table; block A is now ontop of block C, and block C is no longer clear.

In the second elaboration cycle, several things happen with the set of operators:

1. The operator that moved block A to ontop of block C is terminated. This signals to Soar that it can select a new operator.

2. Some of the operators that were previously proposed are retracted because they no longer apply to the state. Five of the six operators are retracted. Retractions are not shown in the trace.
   - the two operators that could move block A are retracted because block A has changed its position.
   - the two operators that could move block C are retracted because block C now has block A ontop of it, and cannot be moved.
   - the operator that moves block B to block C is retracted because block C now has another block ontop of it.

3. Two new move-block operators are proposed: block A may be moved back to the table, or block A may be moved ontop of block B.
Discussion of Blocks-World by Elaboration Cycles

In the third elaboration cycle, both of the move-block operators proposed in the previous elaboration cycle are made indifferent.

There are now three operators for Soar to select from: moving block A to the table or to ontop of block B, and moving block B to ontop of block A.

The second decision cycle ends when Soar (randomly) selects one of these three operators.

The third and fourth decision cycles proceed much the same as the second: the state is updated to reflect the application of the operator; the current operator is terminated; invalid operators are retracted; new operators are proposed; new operators are made indifferent; and one of the proposed operators is randomly selected.

Following the fourth decision cycle, the state is updated. The current operator is terminated, old operators are retracted and new ones are proposed, but at the same time, the Soar program detects that the goal has been achieved. The Soar program sends a signal for Soar to halt. (The fifth decision cycle begins, but Soar halts before it finishes this decision cycle.)

Which Instantiations Retract Their Actions?

To oversimplify:

- Production instantiations that apply operators do not have their actions retracted.

- Most other instantiations will retract the preferences they created when the instantiation is no longer matched in working memory.

This distinguishes two types of productions in Soar, generally called O-supported productions and I-supported productions.

NOTE: The details are a little more complicated, and we'll discuss them later.
Impasses in Problem-Solving

An impasse may arise when Soar is unable to accomplish one of the (P) problem-solving functions (from slide 37):

1. Soar cannot select a new operator because none are proposed.
2. Soar cannot select a new operator because multiple operators are proposed and the comparisons are insufficient to select one of the proposed operators.
3. (does not involve production firings)
4. Soar has selected an operator, but doesn’t know how to apply it.
5. Soar has selected an operator, but doesn’t recognize that it can be terminated.

Note that the impasse happens during the third problem-solving function (selection of the operator), but happens because one of the other four problem-solving functions was incomplete.

Impasses in Problem-Solving

For example, two operators may be proposed as acceptable. But if they are not also mutually indifferent, or one is not better than the other, the comparison problem-solving function has not completed.

In response to an impasse, Soar creates a subgoal; in the subgoal, Soar tries to complete the operator comparison.

Attribute Impasses: Soar can also reach an impasse for non-operator attributes; this is called an attribute impasse, and does not lead to a subgoal. Soar can continue problem solving.

NOTE: We’ll discuss the details of impasses in a later talk.
Summary of Soar

At a coarse level of detail, the Soar architecture is very simple. Here are a few things to keep in mind:

**Soar runs continuously.**

Once started, Soar will continue to execute until the program sends a halt signal or the user interrupts Soar.

**Productions fire and retract.**

All productions that can fire do so, and all productions that can retract do so. (This usually means that changes have been made to preference memory, and that usually means that changes have been made to working memory, so often, this leads another set of productions to fire and retract.)

**Changes to working memory are based on preferences.**

The actions of productions do not make changes to working memory directly. Instead, they create preferences; the Soar architecture evaluates these preferences to determine the contents of working memory.

---

Summary of Soar, part 2

**Soar is always trying to select a new operator.**

When there are no more productions to fire or retract (called quiescence), Soar tries to select a new operator. For this to happen, the current operator must be terminated (with a **reconsider** preference) and there must be at least one operator suggested (with an **acceptable** preference).

**Soar will reach an impasse if it cannot select a new operator.**

If the current operator is not terminated, or if there are no operators suggested, or if there are multiple operators suggested but not enough preferences for Soar to pick one of the operators, Soar will be unable to select a new operator at quiescence. This situation is called an impasse.

**Soar creates a subgoal in response to an impasse.**

If Soar is unable to select a new operator, it creates a subgoal, which is actually the initial state for resolving the impasse. In the subgoal, Soar tries to do problem solving to either propose new operators or determine which proposed operator it should select in the parent state. Solving this problem is a subgoal of whatever task Soar was working on before.
Summary of Soar, part 3

Subgoals are removed when the impasse is resolved.
The new state created in response to an impasse is a temporary state that will go away when the problem is solved.

Learning may occur when an impasse is resolved.
When learning is on, Soar tries to learn something whenever a subproblem is solved and an impasse is resolved. Soar creates a new production that is added to production memory. If Soar is in a similar situation in the future, this production may fire and eliminate the need for a subgoal.
Links in working memory:

- All objects must be linked to a state (or impasse).
- Objects that are no longer linked are automatically removed.

*NOTE:* These two figures illustrate the same working memory using different abstractions.

---

**A Few More Details on Working Memory, part 2**

*Special objects in working memory:*

**states:** Can only be created by the Soar architecture.

**impasses:** Can only be created by the Soar architecture.

**operators:**

1. Decision of operators is deferred until the decision cycle.
2. Acceptable preferences for operators (proposals) appear in working memory (so that productions can test for them).

Soar creates states and impasses automatically, with some predefined attributes.

- Productions may suggest additional attributes for state and impasse objects in working memory.
A Few More Details on Production Memory

Production conditions must be linked
- One condition of each production must refer to a state (or impasse) object.
- All other conditions must be linked (directly or indirectly) to that.
- WME's proposed in the actions of a production must be linked to the conditions.

Example production in pseudo-code

\[
\begin{align*}
\langle \text{state1} \rangle & \text{ is a state} \\
\langle \text{state1} \rangle & \text{ has a thing } \langle \text{block1} \rangle \\
\langle \text{state1} \rangle & \text{ has a thing } \langle \text{block2} \rangle \\
\langle \text{block1} \rangle & \neq \langle \text{block2} \rangle \\
\langle \text{block1} \rangle & \text{ is clear} \\
\langle \text{block2} \rangle & \text{ is clear} \\
& \text{ suggest an operator to move } \langle \text{block1} \rangle \text{ ontop of } \langle \text{block2} \rangle
\end{align*}
\]

\(<x> = \text{variable}\)

A Few More Details on Preference Memory

Preferences are suggestions or imperatives about the objects that should be present in working memory:
- Preferences express relative or absolute merits of a particular WME.
- Preferences may be contradictory (working memory is not).
- Duplicate preferences can exist (duplicate WME's cannot).

Preferences remain until removed for one of several reasons:
1. A preference may be removed when the production that created it retracts.
2. Preferences that are not linked to a state or impasse are automatically removed.
3. A new preference may be created that permanently rejects a proposed WME.
Recap: Soar's Elaboration Cycle & Decision Cycle

Soar's Decision Cycle

Elaboration phase
- multiple elaboration cycles
- operator proposal, comparison, application, termination; also elaboration

Quiescence
- no more productions are ready to fire or retract

Decision phase
- Soar attempts to decide on a new operator
- operator preferences are evaluated
- current operator must be terminated for new to be selected
Soar's Elaboration Cycle

Elaboration cycle:
1. Preference phase
   - productions fire and retract
2. Working Memory phase
   - non-operator preferences are evaluated by the decision procedure

*NOTE:* There seems to be a missing step 3: the match phase. This is not a separate phase because the match set is updated dynamically as changes are made to working memory. That is, this is conceptually a distinct phase, but not literally.

A new elaboration cycle begins as long as there are productions ready to fire or retract. (If not, Soar proceeds to quiescence.)

The decision procedure used in the elaboration cycle (for non-operator preferences) is the same procedure used in the decision phase of the decision cycle.

Illustration of Decision Cycle

Decision Cycle
- Elaboration Phase
- Decision Phase
- Quiescence

Decision 1
- Elaboration Phase
- Decision Phase
- Quiescence

Decision 2
- Elaboration Phase
- Decision Phase
- Quiescence

Decision 3
- Elaboration Phase
- Decision Phase
- Quiescence

Quiescence
- newly instantiated productions fire
- productions that are no longer instantiated are retracted

Decision Phase
- newly instantiated productions fire
- all non-operator preferences are considered
- the preferences are evaluated
- elements are added and deleted from working memory

Decision Phase
- newly instantiated productions fire
- all non-operator preferences are considered
- the preferences are evaluated
- a new operator is selected
- a new state is created