Introduction to Intelligent, Knowledge-Based, and Cognitive Systems

89-674

Mor Vered
http://u.cs.biu.ac.il/~veredm/89-674/
mvered89230@gmail.com
Sunday 10-11, Building 202, Room 119
Soar 9.5.0

- A unified architecture for developing intelligent systems – a programming language embedding in it a specific theory of the appropriate primitives underlying reasoning, learning, planning, and other capabilities that are hypothesized to be necessary for intelligent behavior.

- Our hypothesis is that Soar is appropriate for building autonomous agents that use large bodies of knowledge to generate action in pursuit of goals.

- Provides the fixed computational structures in which knowledge can be encoded and used to produce action in pursuit of goals

- No prior knowledge of Soar or computing programs needed
What is Soar

- A control architecture
- A rule-based language
- An integrated architecture
  - learning, planning, problem solving, …
- An implementation of cognitive theories
  - Lots of human-factors and cognitive science research

  • Modeling complex, dynamic, flexible behavior
Components of Soar: Memory and rules

- Working (short term) memory (known now)
- Rules

\[
\text{IF } \langle A \text{ is true} \rangle \\
\text{THEN } \langle B \text{ be made true} \rangle
\]

- All rules fire in parallel, e.g.,
  - IF \langle being shot \rangle, THEN \langle Want to Run \rangle
  - IF \langle being shot \rangle, THEN \langle Want to Fight \rangle

  - Both \textit{Want-Run} and \textit{Want-Fight} will be true
Components of Soar: Operators

- Rules encode long-term memory
- Rules organized into groups (Operators)
  - Propose, initialize, select, and terminate operators
- Behavior controlled by (de-)selecting operators
  - Each operator encodes some atomic unit of behavior
  - Sequences of operators generate behavior
- Allow mix goal-directed and reactive behavior
Components of Soar: Impasses

- Operators encode task-solving knowledge
- Compete for control of agent
- When knowledge insufficient: Impasse
  - Conflict detected, no operator appropriate
- Impasses trigger problem-solving, reasoning
- This gives Soar:
  - Flexibility (no commitment to predefined sequence)
  - Ability to tackle new situations
Soar use in building agents

- Multiple projects use Soar as backbone
  - Since 1993
  - Industrial and academic research projects
- IFOR project:
  - Synthetic FWA and RWA pilots
  - Coordinated missions, focus-of-attention
  - Generally considered huge success by DARPA
- SoarTech spin-off of successful IFOR project
  - Other spin-offs, other research projects
Projects using Soar

- IFOR, mission rehearsal, mobile infantry
- RoboCup soccer (virtual players), game environments
- Distributed decision-support applications
Why Soar for building agents

- Situational awareness
  - Rules, global access to working memory
- Controlled, deterministic behavior
- Flexible, dynamic response
  - Opportunism, responsiveness to novel situations
- Significant know-how exists
  - Both academic and industrial
- Learning (mostly disabled in virtual environments)
Visual Soar

• Soar has its own editor, called VisualSoar, which we highly recommend for use in developing Soar programs. VisualSoar is part of the standard installation and is also available from the Soar homepage.
High Level View Of Soar

Rules encode long term knowledge in Soar. We build an agent by creating rules.

Working memory is the agents’ knowledge. Elements are created by rules and sensors. Core knowledge created automatically.

Working memory is matched against rule memory to determine which rules will fire. When rules fire, they can change working memory and perform simple actions such as printing messages in the interaction window.

Underlying architectural processes (select operators, create states, learn new rules). Not changeable.
The symbol manipulation level

- **Working Memory:**
  - Set of Working Memory Elements (WMEs)
  - “things that are known”

- **Productions:**
  - If/Then Rules
  - “Test and modify WMEs”

- Productions match working memory
- Productions modify working memory (when matching)
Identifiers can have links emanating from them. Constants cannot.
The links are called attributes and are prefaced by a “^”.
Only identifiers have attributes.
Working Memory

• A working memory object usually represents something about the world, such as a block, a wall, a piece of food, or a cell on the board.

• Working memory usually also contains objects that are only conceptual things and do not have an identifiable physical existence, such as state s1, which organizes other objects, relations, and properties.

• A collection of working memory elements that share the same first identifier is called an object.

• The working memory elements that make up an object are called augmentations.

• The working memory elements for the blocks would be as follows:
Working memory

- Working memory contains all of a Soar agent’s dynamic information about its world and its internal reasoning (sensor data, intermediate calculations, current operators and goals).

  - Soar’s working memory is a huge graph

- Working Memory Elements (WMEs)

  (S1 ^io I1)
  (I1 ^input I2)
  (I1 ^output O1)
  (I2 ^left-light on)
  (I2 ^right-light off)

- A Few WMEs are special

- Minimal amount of working memory
  - Connected to sensors and actuators (I1,I2,O1, ...)
  - Roots for the graph (S1)
Soar VS. Other Programming Languages

- In contrast to C or Java, which have programs that are executed sequentially, Soar has rules that are matched and executed in parallel.
- C and Java have data structures, variables, etc. while Soar has working memory.
- While these other languages have a combination of local and global data, Soar has no local data – everything in working memory is global.
Installation

• [http://soar.eecs.umich.edu/articles/downloads/soar-suite/105-soar-tutorial-9-4-0](http://soar.eecs.umich.edu/articles/downloads/soar-suite/105-soar-tutorial-9-4-0)

• Download

• Double Click SoarJavaDebugger.bat

• Default agent ‘soar1’ is created.
Soar Debugger

Commands to run, single step or stop an agent

Removes all existing rules from memory and initializes Soar
Hello World Rule

• In order to execute the hello-world rule, it must be loaded from a file into Soar’s rule memory (...\SoarTutorial_9.4.0-\Windows_64bit\Agents\hello-world-rule\hello-world-rule.soar)

• When rules are loaded into Soar, an “*” is printed for each rule. When the file has loaded, you will see the word “source” followed by the location of the file in the interaction window
Start executing Soar by typing run in the command window or clicking on the run button.

Trace of run command being executed.

"Hello World" printed out by rule followed by halt.

Soar always automatically creates a state (S1) for all data.
I think, therefore I exist?

- Our first agent:
  - Write “Hello, World!” to the screen

- Procedurally, we just tell it to do it:
  - C: { printf(“Hello, World!\n”); }  
  - C++: { cout << “Hello, World!”;  }
  - BASIC: 10 PRINT “Hello, World!”
  - PASCAL: begin writeln(“Hello, World!”); end.

- ….

- How do we do it in Soar?
I exist, therefore ....

- Soar is **not** procedural
- We need to think about this in IF/THEN form

I exist, 

Therefore

I say hello.
I exist, therefore ....

- Soar is **not** procedural
- We need to think about this in IF/THEN form

- **IF**
  - There is a state

- **THEN**
  - Write “Hello, World!”
  - Stop
I exist, therefore ....

- Soar is not procedural
- We need to think about this in IF/THEN form

```plaintext
sp {{ hello-world*1
    (state <s> ^type state)

☐ THEN
- Write “Hello, World!”
- Stop
```
I exist, therefore ...

- Soar is **not** procedural
- We need to think about this in IF/THEN form

```plaintext
sp { hello-world*1
    (state <s> ^type state)
  -->
    (write |Hello, World!|)
    (halt)
}
```
Soar Syntax

sp stands for “Soar production” and starts every rule.

The “{“ starts body of a rule.

This is the if part of the rule (condition).

The --> separates the if and then parts of the rule.

The name of the rule immediately follows the “{“.
The name can include letters, numbers, “-“, and “*”.

These are the then parts of the rule (actions).

The “}” ends the rule.

There may by several conditions and several actions :

sp { rule*name
   (condition)
   (condition)
   -->
   (action)
   (action)
}
Soar Syntax

• Following the name, there must be at least one condition.
• A name cannot be a single letter followed by a number (S1,O45).
• A condition tests for the existence (or absence) of data in working memory. If all of the conditions match, then the rule will fire and all of the actions are performed.
• Most actions create new working memory elements or remove existing working memory elements, while others perform functions such as writing text to the screen or halting as in hello-world.
Soar Syntax - (state <s> ^type state)

• When every Soar agent is created, it has (s1 ^type state) in working memory – signifies the agent exists (s1 is just an arbitrary symbol – if soar was implemented differently could have gotten a different name each time, so mustn’t rely on the name).

• Therefore we need a test that there is an identifier, but without testing a specific value. We will use a variable.

• A variable is a symbol surrounded by “<” and “>”, such as <s>

• in Soar, the first condition of every rule must start with the word “state”.

• (state <s> ^type state)
Tricky....

- Why are we testing (state <s> ^type state)?
  - Why not test (state <s>)?

```plaintext
sp { hello-world*1
    (state <s> ^type state)
  -->
    (write |Hello, World!|)
    (halt)
}
```
Tricky....

- Why are we testing (state <s> ^io <i>)?
  - Why not test (state <s>)

- Because:
  - All rules match all WMEs.
  - State contains:
    - (S1 ^io I1), (S1 ^type state)
  - So a condition (state <s>) will match twice
    - Same as matching (state <s> ^<attr> <v>)
Soar Syntax - (write |Hello World|)

- Vertical bars, “|”, mark constants with special characters.
- These constants can contain any characters, which allows you to print spaces, upper and lower-case letters.

Soar Syntax - (halt)

- Special command in Soar that halts the agent.
Tricky....

- Why (halt)? Why are we stopping Soar?

sp { hello-world*1
    (state <s> ^type state)
  -->
    (write |Hello, World!|)
    (halt)
}
Tricky....

- Why (halt)? Why are we stopping Soar?
  - Because we wrote a program, not an agent.
  
- No sense-think-act cycle here
- Soar is intended to build agents
- Loops through decision cycles

  - What would happen if there was no (halt) command? Try it at home...
Firing rules in a cycle

- Rules fire (once!) for every new match
- So first, fire because of new inputs
- But then, these change WMEs
  - If  (input ^light *on*)
  - Then  (<s> ^saw-the-light true)
- So rules can fire based on changed WMEs
  - If  (state <s> ^saw-the-light true)
  - Then  (state <s> ^save-me *yes*)
Elaborations

- This constant firing of rules continues
- Results stay as long as rule’s conditions valid
  - If rule no longer matches, results retracted
- Continues until:
  - Quiescence (no new WMEs generated)
  - max-elaborations reached (a parameter)
- Elaborations: rules with no permanence
Infinite loop: The obvious version

- In Soar it is possible to test for the absence of working memory elements by preceding the attribute with “-“.
- Every rule in Soar must have at least one positive condition, so for testing empty situations you still need to test that the state exists and in this case we use (^superstate nil).
- (state <s> ^superstate nil)
  -(<s> ^name)
Infinite loop: The obvious version

\[
\begin{align*}
\text{sp} \{ & \text{hello-world*1} \\
& (\text{state } <s> \ ^\text{type state}) \\
& -(<s> \ ^\text{tested } ^\text{yes*}) \\
& \rightarrow \\
& (<s> \ ^\text{tested } ^\text{yes*}) \\
\}
\end{align*}
\]

\[
\begin{align*}
\text{sp} \{ & \text{hello-world*2} \\
& (\text{state } <s> \ ^\text{type state}) \\
& (<s> \ ^\text{tested } ^\text{yes*}) \\
& \rightarrow \\
& (<s> \ ^\text{tested } ^\text{yes*} - ) \\
\}
\end{align*}
\]

- First hello-world*1 fires, adding ^tested
- Then hello-world*2 fires.
- This causes hello-world*1 to fire again...
  - Until max elaborations reached
I-Support

• *Non-operator application rules* include rules that propose an operator, rules that compare operators, rules that elaborate operators, or rules that elaborate the state.

• Working memory elements created by these rules are removed from working memory if the rule no longer matches.

• Meaning rules that automatically retract their results when no longer match.

• These are said to have *instantiation-support* or *i-support*, meaning they will persist only as long as the rule instantiation that created them still matches.
i-support

- WMEs that have no permanence
- Called *i-supported*
- Are used to make automated inferences

- For instance, when target changes location:
  - Re-compute bearing
  - Re-assess threat according to various rules
  - Re-compute possible interception points
  - Re-think optimal weapons use
Infinite loop: The tricky version

\[
\text{sp \{} \text{ hello-world}^*1
\hspace{0.5em} (\text{state} \text{ <s> } \text{^type state})
\hspace{0.5em} -(<\text{s}> \text{^tested} \text{*yes*})
\hspace{0.5em} -->
\hspace{0.5em} (<\text{s}> \text{^tested} \text{*yes*})
\}\n\]

- First hello-world*1 fires, adding ^tested
- Then hello-world*1 is unmatched, removing.
- This causes hello-world*1 to fire again...
  - Until max elaborations reached
Homework

• Download Soar (please bring laptop to next lesson as we will be solving a problem together)

• Read “Jump Start To Soar Programming” from website