Soar

What is Soar?

- State-Operator-And-Result
- A candidate unified theory of cognition embodied in a cognitive architecture
- Developed by John Laird, Paul Rosenbloom and Allen Newell starting in 1982 at Carnegie Mellon University
- Ostensibly a rule-based model of human problem-solving at the knowledge level

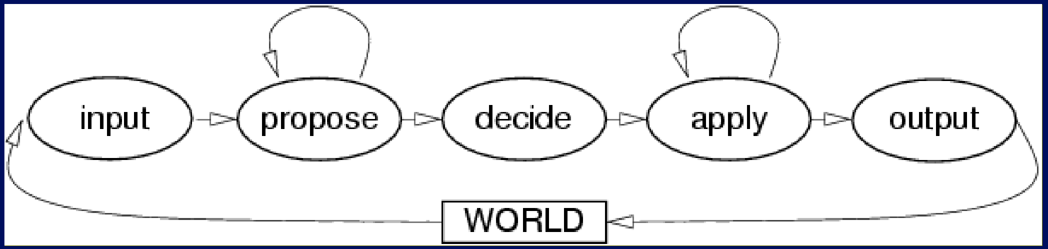
Soar Architecture Overview

- **Long-term knowledge** is encoded as *“if-then”* production rules which match against and update short-term memory
- **Short-term memory** is represented as a connected graph of objects containing attribute-value pairs
- Most primitive features of an AI system are addressed including automatic **sub-goaling, belief maintenance, decision making,** and **symbolic learning.**
- Other features such as forward planning may be implemented within the Soar language
- Interaction with the external environment is memory-mapped and amounts to *“action as command”*

QinetiQ

Soar Decision Cycle

- The execution of a Soar agent is described by the Soar decision cycle:



- The **propose** phase suggests **operators** (things to do), a **fixed decision** function chooses between them, then an **apply** phase applies the operator and then the cycle restarts
- Within the **propose** and **apply** phases production rules match and fire in **lock-step synchrony** until no more rules may fire and **acquiescence** is achieved
- The model of control, i.e. how rules are matched and fire, is important as in general there will be **race conditions** between competing production rules

QinetiQ

Analysing Soar Agents

- We use a process of translation to a formal framework, followed by analysis
- Our formal framework uses CSP and model checking to perform static analysis of Soar programs
- Analysis covers general healthiness properties, akin to exception analysis of a conventional program
- Specific properties such as the non-occurrence of dangerous events may be checked

Healthiness Properties

- Essentially the absence of Soar behaviours that would be considered *"bad"* in any context
- Interested in bad behaviour caused by **incomplete knowledge, inconsistent world-views, erroneous production rules, rule interactions** and so on
- Bad behaviour in Soar may include **livelock, unintended impasses** and **illegal actions**
- **Redundancy** of production rules is also undesirable and may also be checked for

Livelock (over decisions)

- here a Soar agent performs an **infinite sequence** of elaborations; productions fire continuously and the agent never reaches the next decision
- many causes may be eliminated through judicious **constraints** on the Soar language
- **enabling-disabling** forms of livelock are immediately detectable in CSP
- other forms of livelock such as counting require techniques such as placing **finite bounds** on *potentially* infinite sets

Ilegal Actions

- here a Soar agent issues an actuator command that *cannot* or *should not* be actuated
- most simulations issue a warning and simply ignore the action
- in reality such illegal actions may not be ignorable and probably highlight problems with the agent itself
- illegal actions may be detected given a suitable model of the environment to use as a watch-dog process

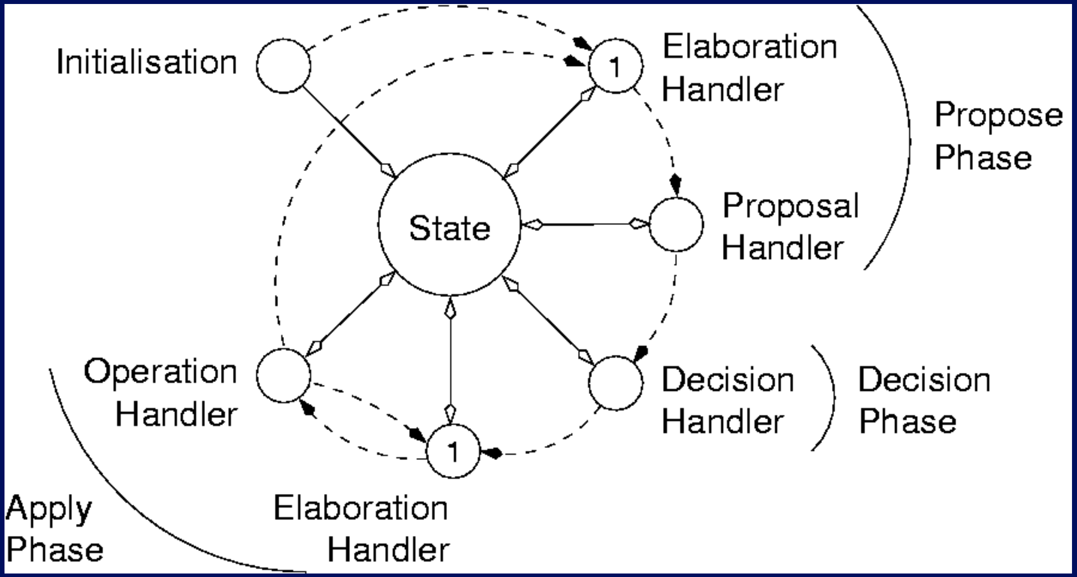
QinetiQ

CSP Model of Soar

- Our CSP model of Soar starts with the concept of a *“datamap”*, a static vision of all possible objects, attributes and values that may exist for a given problem space
- We model Soar as a series of inference engines manipulating such datamaps
- A particular agent is provided as *data* to a generic CSP model of Soar
- Data takes the form of simple firing rules produced by automatic translation from the original production rules, as well as type definitions describing the possible values of all object attributes

CSP Model of Soar (2)

- The model is actually composed of three decoupled inference engines, a decision function, (persistent) state process, and input/output processes (not shown)



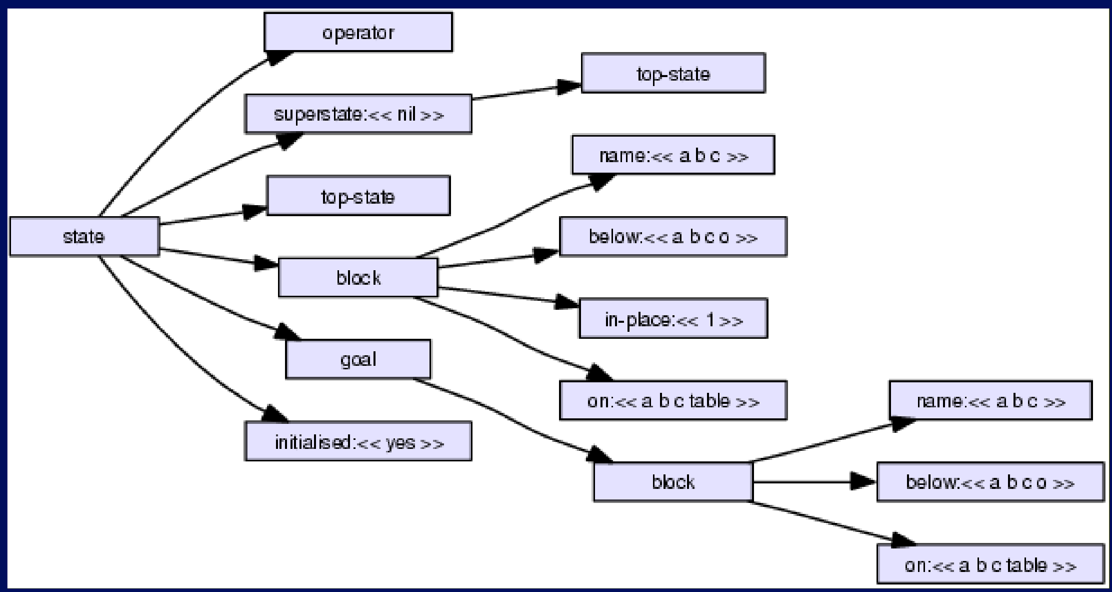
- The dotted lines represent mode changes (only one inference engine is enabled at any one time), while the solid lines represent get/set operations on the persistent state

CSP Model of Soar (3)

- Mode changes roughly correspond to the match-fire-update cycles within the propose and apply phases
- Separation of the types of production rule improves the clarity of the model, but also improves the efficiency of the CSP analysis
- Decoupling of the inference engines is only sound through constraints on the Soar language

Soar Datamaps

- Our analysis of Soar currently expands each production rule to a number simple firing rules over atomic facts
- As production rules match by *unification* the analyst must provide a description of **structure** as well as data types
- To do this we use a Soar *"datamap"*, a map describing the possible links between objects and possible values for simple attributes



Current Status of the Model

- The current CSP model is capable of detecting all of the healthiness properties
- However:
  - the model does not yet support certain key features of Soar including sub-goaling, and objects with multiple parents
  - Detection of "operator-no-change" impasses may currently raise *false-negatives* (impasses that do not exist)
  - Translation is not quite complete and may require changes to the model
- Sub-goals will form the basis for a decomposition of the CSP analysis
- Multiple parents will be supported but their use sensibly constrained

Soar Language Constraints (2)

- Example fundamental constraints:
  - Belief-maintained rules may not contain negative actions
  - An object/attribute may be created/removed by belief-maintained rules or a normal rules but not both
  - Attribute tests may not use unbound variables
  - Operator proposals must specify all of their arguments
- Example current limitations:
  - Production rules may not contain negated conjunctions
  - Datamaps and productions may not contain cycles or many-to-one links

Soar2Csp (a prototype translator)

- the translator is responsible for the static expansion of Soar productionibo CSP firing p

  nputibo a modelncheckhr foranalyns/sp
- nterlct wich theworOld theanalynt munt alp

- ch thegenerticCSP modeln of Soa, theoutpute foms a p

Soar2Csp (2)

- Features:
  - A complete LALR Soar language parser/simplifier
  - Automatic partition of the datamap

Example Translation

- A Soar Blocks World production rule:

production name

conditions

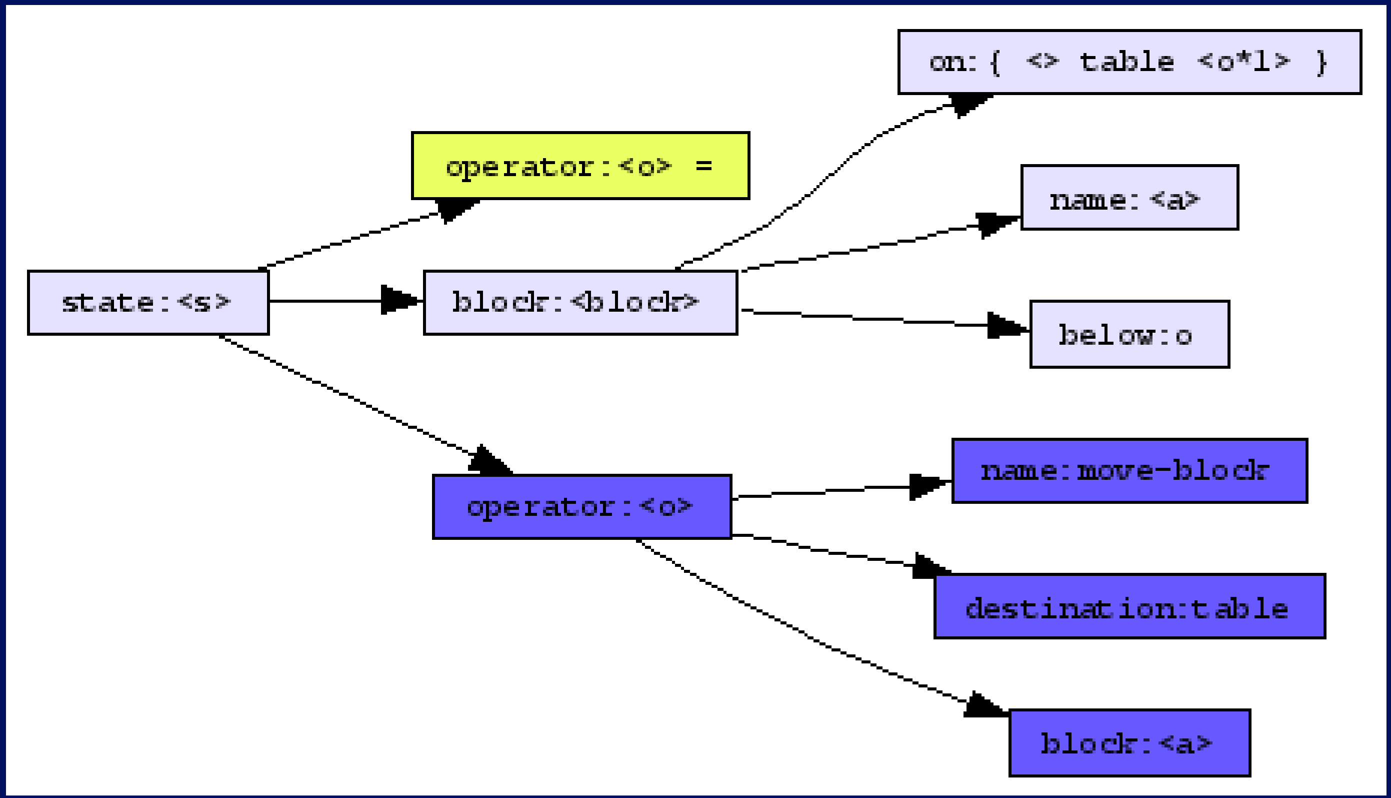
actions

operator name

operator proposal

Example Translation(2)

- The parsed production graph:



Example Translation (3)

- The CSP firing rules (as a set comprehension):

production name

conditions

operator proposal

variable generators and constraints

Current State of the Translator

- The translator works well and has now been validated in a CSP analysis of a real UAV Soar agent
- This has revealed a few out-standing issues with the translation:
  - Value tests in negated conditions are ignored
  - Multi-attribute actions are translated (and perhaps modelled) incorrectly
  - Operator application productions may serve multiple operators, but translate to only one operator
  - Garbage collection in Soar (i.e. recursive delete) is ignored
- Many issues were addressed during the validation
- Out-standing issues are currently handled by hand changes to the CSP generated

Conclusions

- We have identified concrete healthiness properties of any Soar agent
- We have identified Soar language constraints that will be useful for modelling and analysis
- We have made significant progress towards analysing Soar agents
- Both the formal CSP model of Soar and the Soar to CSP translator require improvements

QinetiQ

Questions and Answers