1 Top level design

Figure 1 presents the top level design for the CSP model of, a heavilly constrained version of, Soar [?], where the four Soar phases are broken down into seven main processes, which communicate via a shared state.

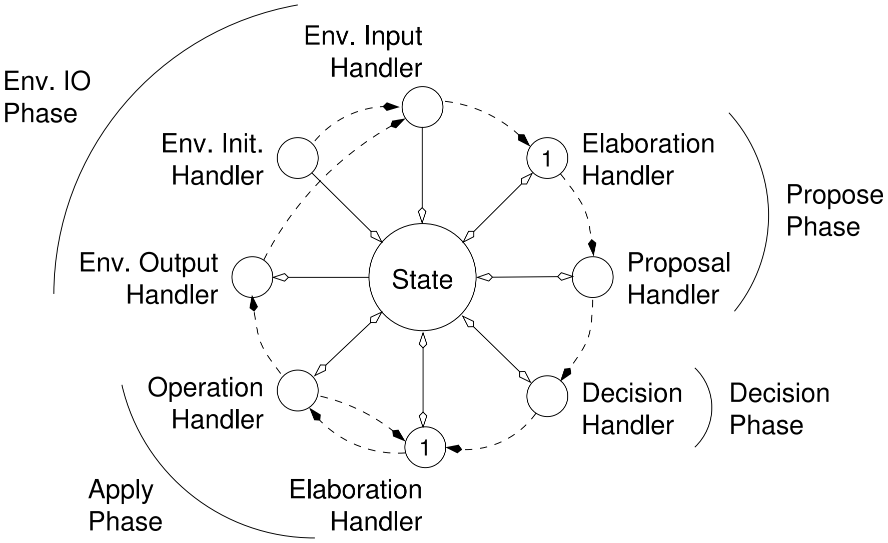

Figure 1: The top level design overview – with Soar phases

The reaminder of this section contains ...

1.1 Phases of Control

The Soar language conceptually has three main phases of control:

propose phase – for establishing which "operations" can be applied;

decide phase – for choosing the "operation" to apply;

apply phase – for executing the chosen "operation".

In addition to the three main phases of control, there is also environmental IO phase, which enables Soar programs to interact with their environments. From the modelling perspective the environmental phase is split into three parts, initialisation, input and output. Thus, the CSP model of Soar has six phases, which are modelled by the following CSP datatype:

```
datatype Phase = init_p       -- { start,                                stop }
               | env_input_p  -- { start,                                stop }
               | propose_p    -- { start, elaborate, perform,            stop }
               | decide_p     -- { start,                                stop }
               | apply_p      -- { start, elaborate, perform, iterate, stop }
               | env_output_p -- { start,                                stop }
```

Each of these phase can be "started" and "stopped"; the propose and apply phases also contain internal moding, which is worth controlling at this level. In the case of the propose phase, the controller coordinates two inference engines: one for elaborating all observations that can be made about the current state of the CSP model of the Soar program; and the other inference engine for determining which proposal rules could apply, and there relative preferences. The apply phase is similar to the proposal phase, except that the second inference engine actually determines how state should be changed; having performed the changes, the apply stage is "iterated", until no further changes are required. Note that this can lead to an infinite number of iterations within the apply phase, such as when the apply phase keeps swapping the contents of two state variables.

> **Aside 1.1:** In Soar the propose phase has an "iterative" nature; our restrictions limit these to only positive conclusions, thus the whole iterative cycle can be coped with by a single inference engine. So why does the CSP model contain two inference engines within the propose phase? There are actually several reasons: First some rules (productions) are shared between the proposal and application phases; in particular, those rules that can observe "facts" about the current state. Instead of having a copy of these rules in both phases inference engines, these inferences can be validly pre-computed, by another *common* inference engine.
>
> Second, being able to split the inference engine in two, is arguably more efficient; as there are less potential permutations of "firing" (deduction) events. However, this reason is slightly dubious, as the implementation technique essentially linearises both the inference engine computation phase, and the communications into and out of state.
>
> Third, being able to classify each soar production as either an elaboration (for observations), a proposal, or an operator (for application phase), provides a useful conceptual separation of concerns. And it makes it difficult to "hack" in productions that cross these conceptual boundaries, and thus violate the constraints that we have placed on the Soar language.

From a modelling perspective the phase of control is determined by events that record both the phases name and mode.

```
datatype PhaseMode = start | elaborate | perform | iterate | stop

channel phase:Phase.PhaseMode
```

These phases are coordinated by a phase controller, which starts by selecting the initialisation phase. This phase may be used to load specific initial state configurations into memory, though care must be taken to ensure that it coordinates with the environment in an appropriate manner. The following CSP formally models the initialisation phase by delimiting the phase by the appropriate start and stop events, then proceeding to the environmental input phase.

```
PHASE_COORD_INIT = phase.init_p.start ->
                   phase.init_p.stop ->
                   PHASE_COORD_INPUT
```

The environmental input phase provides an opportunity for the environment to update the state. It is delimited by start and stop events, before continuing onto the propose phase.

```
PHASE_COORD_INPUT = phase.env_input_p.start ->
                    phase.env_input_p.stop ->
                    PHASE_COORD_PROPOSE
```

The propose phase is slightly more interesting, as events are provided for the purposes on managing its significant activities. In particular:

- The "`....elaborate`" event triggers the activation of the elaboration inference engine, along with its custom built state interaction coordination process. Note that this inference engine is modelled by the `ELAB_IE` process that can be found in the "soar/elab_ie.csp" file.

- The "`....perform`" event triggers the activation of the proposal inference engine, along with its custom built state interaction coordination process. Note that this inference engine is modelled by the `PROP_IE` process that can be found in the "soar/prop_ie.csp" file.

```
PHASE_COORD_PROPOSE = phase.propose_p.start ->
                      phase.propose_p.elaborate ->
                      phase.propose_p.perform ->
                      phase.propose_p.stop ->
                      PHASE_COORD_DECIDE
```

The decide phase is modelled by start and stop events, which coordinate with a customised variant of the Soar decision algorithm. This algorithm caters for only a subset of the Soar preferences, but for those preferences it mirrors the Soar algorithm's behaviour.

```
PHASE_COORD_DECIDE = phase.decide_p.start ->
                     phase.decide_p.stop ->
                     PHASE_COORD_APPLY
```

The application phase is similar to the proposal phase, except: that an application inference engine is run instead of a proposal inference engine; and that the elaboration-application cycles until there are no further persistent state updates to perform. Note that this underpinned by the inference engine's ability to determine whether any new "fact" (updates) have been learnt, as denoted by the ...`exists.`*b* event, where *b* is either `true` or `false`.

```
PHASE_COORD_APPLY = phase.apply_p.start -> PHASE_COORD_APPLY_LOOP

PHASE_COORD_APPLY_LOOP = phase.apply_p.elaborate ->
                           phase.apply_p.perform -> (
                             phase.apply_p.iterate -> PHASE_COORD_APPLY_LOOP
                             []
                             phase.apply_p.stop -> PHASE_COORD_OUTPUT
                           )
```

Having completed the apply phase, the externally visible areas of persistant state are sent to the environment for processing. The process of sending the state to the environment is delimited by the start and stop output phase events as usual.

```
PHASE_COORD_OUTPUT = phase.env_output_p.start ->
                     phase.env_output_p.stop ->
                     PHASE_COORD_INPUT
```

1.2 The modelling of State

Soar has several different types (areas) of memory. From our modelling perspective two types of memory are of particular interest:

working memory – essentially Soar's main memory;

preference memory – a special memory for storing proposals and their relative preferences.

In addition to these two types of memory the model introduces a third type for storing which "operator" was chosen during the decide phase. Note that this information could be passed around with the control flow, but it is more convenient to model it as part of the state.

Aside 1.2: Adding a special type of memory to record the last decision made is straightforward and does not add significantly to the complexity of the state (as a whole). It simplifies the modelling of control flow, as all data required by each process within the main control cycle is contained in the common state model (process). This enables the scheduling of the main control cycle process to be done without considering the data that they require.