Interpreted Declarative Representations of Task Knowledge

June 21, 2012

Randolph M. Jones, PhD

The Problem

- Knowledge engineering/Behavior modeling is costly
  - Much of this cost lies in the design, encoding, and debugging of models/programs
- Iterative Soar model development becomes increasingly expensive as the models grow
  - Knowledge is encoded at a low-level where the details overwhelm the modeler's/engineer's ability to understand and maintain the model
- Reuse of models and model components is still rare
  - It is challenging to reuse behavior elements from one application in another within the same architecture
  - Reuse across architectures is nearly impossible

Advantages of a High-Level Representation

- Software engineering has demonstrated improved coding efficiency from languages that decrease lines of code, increase encapsulation, and decrease complexity/branching
- Design at the representation level, hide implementation details
  - Free modeler from architecture-level details
  - Emphasize understandability, maintainability, and reuse
- Prior research with HLSR demonstrated decreased design-to-coding times for novice modelers and significant reduction in code size and complexity
- Opportunities for non-engineers to configure high-level code/parameters

Advantages of a Canonical Representation

- It would be beneficial to increase the ease the creation of reusable representations
  - Higher levels of abstraction have wider reuse potential
- Reuse across behavior models
  - Using goal and knowledge representations in multiple execution agents
- Reuse across applications/model types
  - A single knowledge base for planning and execution agents
  - A single knowledge base for execution and explanation agents
  - A single knowledge base for different architectures/engines
  - Differences between planning, execution, and explanation can be embedded in the interpreter or as add-on knowledge

Interpreters vs. Compilers

- A compiler must be complete before you can use it
  - Difficult to experiment with, add, or change language features
  - Difficult to track down bugs if the compiler is not mature
- An interpreter allows simultaneous prototyping and development of the language and the models that are specified in the models
  - Only need to implement those language features that a model actually uses
  - Easier to add, change, and debug language features
  - Code translation is faster and "just-in-time"
    - Compilers take longer to translate, but provide the opportunity to generate much more efficient code
  - Easier to write interpreters "piece-meal" for different architectures
  - Easier to build alternative interpreters for portions of the language
- Ultimate goal should be a compiler generating efficient code for a fixed, formal language
- OR, can chunking be the compiler?

Declarative Goal Representation

- Goal definitions explicitly specify local and global information to be accessed
- Two types of subgoals:
  - Achieve: Remove subgoal as soon as it is achieved once, or if it becomes "deactivated"
  - Maintain: Remove subgoal only if it becomes "deactivated"
- Automatic binding of parameters across supergoal/subgoal
- Strong typing and error checking available if desired
- Declarative representation of subgoal-activation and goal-achievement conditions
  - Using abstract features that are implemented in domain-specific Soar rules
- Query system ensures that elaborations/computations occur only when something is ready to "consume" them
  - Activation conditions, achievement conditions, choice conditions

Goal Representation Examples (Soar-ified XML)

```
^goal
   ^name fly-flight-plan
      ^parameter
         ^name current-point
         ^global-name current-point
         ^category mission
      ^parameter
         ^name arrived-at-point
         ^property-name arrived-at-point
         ^property-object current-point
```

Goal Representation Examples (Soar-ified XML)

```
^goal
   ^name fly-flight-plan
      ^achieve
         ^name fly-control-route
         ^activate-when
            ^not-equal
               ^parameter-value arrived-at-point
               ^value true
      ^achieve
         ^name fly-control-point
         ^activate-when
            ^equal
               ^parameter-value arrived-at-point
               ^value true
```

Goal Representation Examples (Soar-ified XML)

```
^goal
   ^name fly-control-route
      ^parameter
         ^name current-point
         ^global-name current-point
         ^category mission
      ^parameter
         ^name arrived-at-point
         ^property-name arrived-at-point
         ^property-object current-point
      ^parameter
         ^name waypoint
         ^value
            ^get-waypoint-by-name current-point
```

Goal Representation Examples (Soar-ified XML)

```
^goal
  ^name fly-control-route
  ^maintain
     ^name waypoint-computer-programmed
     ^bind-input
        ^parameter waypoint
        ^subgoal waypoint
     ^activate-when true
  ^maintain
    ^name maintain-group-heading
     ^bind-input
        ^parameter waypoint
        ^subgoal waypoint
     ^activate-when
        ^achieved-subgoal waypoint-computer-programmed
```

Goal Representation Examples (Soar-ified XML)

```
^goal
  ^achieved-when
     ^equal
        ^parameter-value arrived-at-point
        ^value true
```

Other Declarative Information (Soar-ified XML)

```
^formation
   ^type bearing
   ^sub-type defensive
   ^size 4
   ^sub-formation
      ^type bearing
      ^sub-type defensive
      ^size 2
      ^lead lead
      ^wingman wingman
      ^wingman-side
         ^opposite second-lead
```

```
^sub-formation
   ^type bearing
   ^sub-type defensive
   ^size 2
   ^lead lead
   ^wingman second-lead
^sub-formation
   ^type bearing
   ^sub-type defensive
   ^size 2
   ^lead second-lead
   ^wingman second-wingman
   ^wingman-side
      ^same second-lead
```

Next steps

- Continue developing and formalizing declarative representation, together with execution-agent interpreter (three projects)
- Develop explanation-agent interpreter (one project)
- Develop planning-agent interpreter (one project)
- Investigate mapping to alternative declarative behavior representations (one project)
- Develop interpreter that stores declarative representations in semantic memory instead of working memory (internal R&D)
  - Determine whether this is actually useful from performance and learning perspectives

Summary

- Nuggets
  - Representation is working and being used in multiple projects
  - Allows the model builder to focus on higher level abstractions and error checking, independent of production/operator-level details
  - Interpreter eases active, rapid development of the language
- Coal
  - Haven't yet built a number of interpreters we want to try
    - UM-Style interpreter
    - Explanation-agent interpreter
    - Planning interpreter
  - Not sure where to put the representation
    - Intuition is that working memory is the wrong place
    - Future work will evaluate working memory vs. semantic memory
    - May still want to use a compiler in the long run