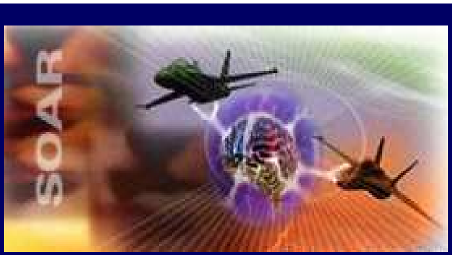
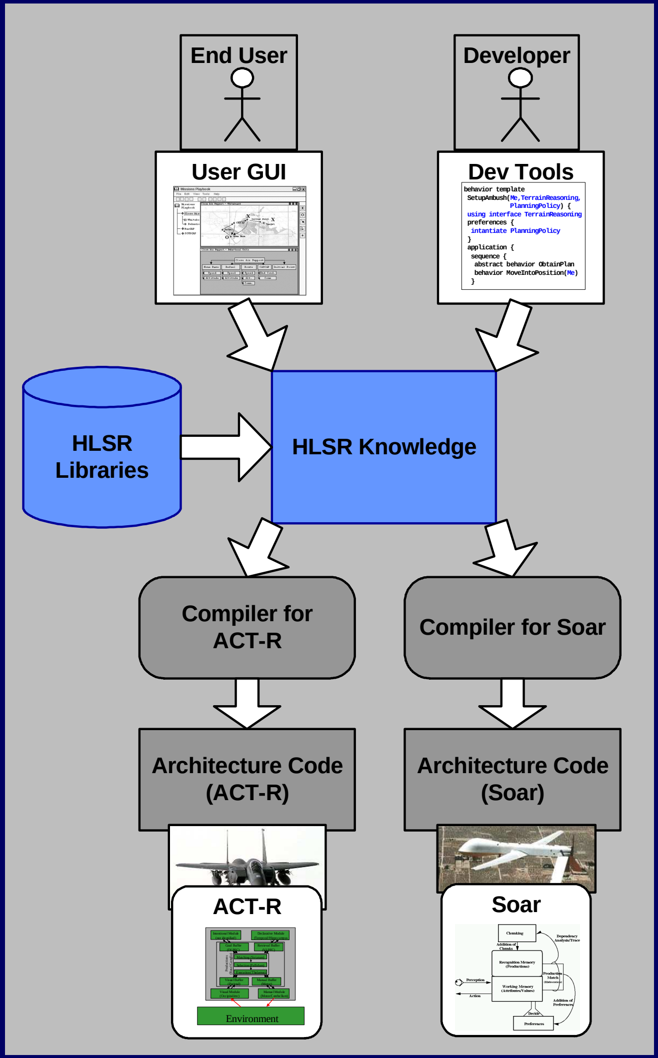


HLSR: Compiling to Soar

Randolph M. Jones
Jacob A. Crossman
Christian Lebiere
Bradley J. Best

What is HLSR?

- High Level Symbolic Representation
- A language for encoding knowledge
- The language is:
  - Architecture-neutral
  - Domain-independent
  - High-level
  - Designed to support reuse
- Target users:
  - Cognitive modelers
  - End user tool developers



```
behavior template
 SetupAmbush(Me,TerrainReasoning,
             PlanningPolicy) {
 using interface TerrainReasoning
 preferences {
  intantiate PlanningPolicy
 }
 application {
  sequence {
   abstract behavior ObtainPlan
   behavior MoveIntoPosition(Me)
  }
```

What HLSR Contributes

- Design at the representation level, hide implementation details
  - Free modeler from architecture-level details
  - Emphasize understandability, maintainability, and reuse
- Why an abstract language?
  - Better tools are necessary but not sufficient
  - Cognitive architectures are necessary but not sufficient
- A language allows merging of different architectural concepts while abstracting low-level details

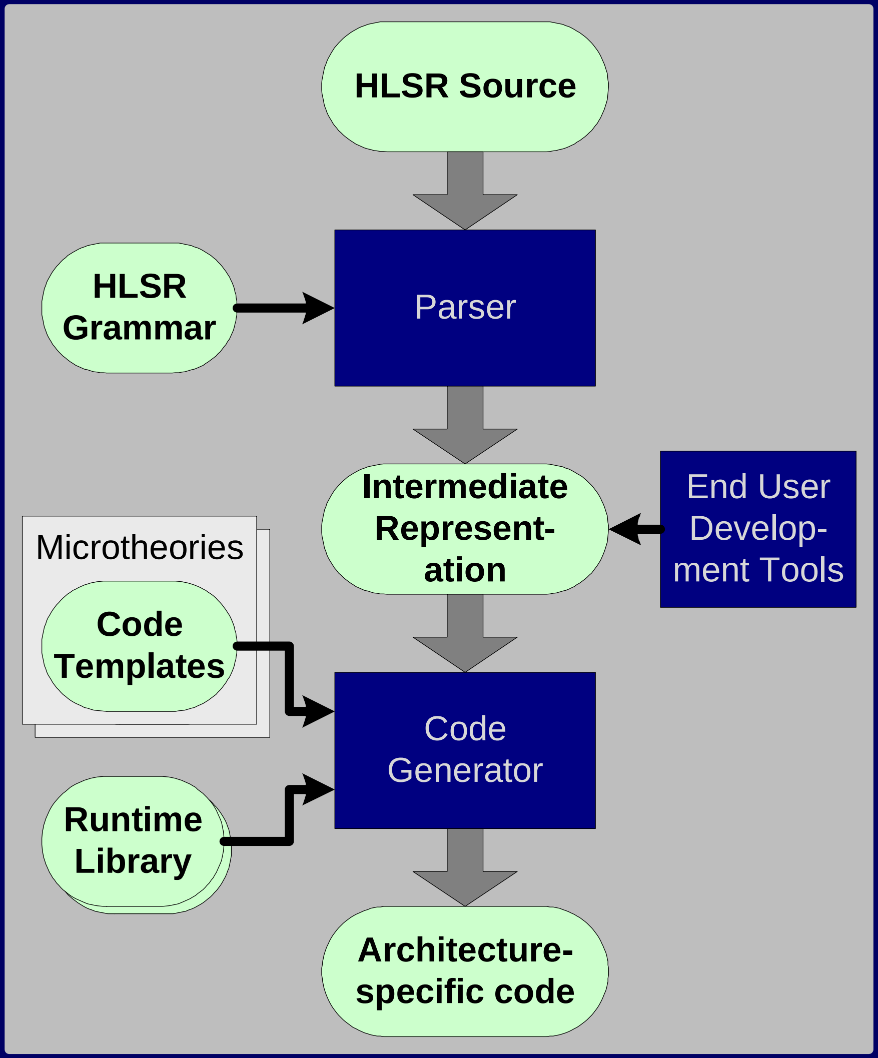
Common Mechanisms Captured by HLSR

- Goals
- Declarative memory
  - Structure and retrieval
- Timely reaction to external events
- Decision processes
  - Goal selection
  - Action selection
- HLSR creates higher-level constructs that map onto different lower-level constructs in different cognitive architectures
  - Compiler should be able to translate HLSR to ACT-R or Soar

Micro-Theories

- Description of structures, templates, and execution strategies used to execute HLSR constructs
    - Architecture-specific
    - Invisible to HLSR developer
- Micro-theories are modular
    - One micro-theory for each HLSR construct

The HLSR Compiler



Abstracting Low-Level Details

- Process tagging
- Integrating knowledge from different models
- Computing answers vs. retrieving stored answers
- Iteration
- Copying
- Complex logic
- Representing sensory-motor interactions

HLSR Building Blocks (Primitive Constructs)

- **Relations**
  - Declarative memory, goals
  - Form: production or rule
- **Transforms**
  - Procedural knowledge
  - Form: body of execution
- **Activation Tables**
  - Pattern recognition for response selection
  - Form: decision matrix

Relation

- A relationship between symbols in declarative memory
- Defined by:
  - Name
  - Attributes
  - Met condition (optional)
- Can be:
  - A fact
  - A goal
  - A request to retrieve something from declarative memory

```
relation Square
  name is a string
  size is a integer

relation SmallerThan
  a is a Square
  b is a Square
  met condition
    a.size < b.size
```

Transform

- A conditionally executed procedure
- Defined by:
  - Name
  - Trigger conditions
  - Body (set of actions)
- Actions execute serially
- Multiple transforms may execute in parallel
- Failure to execute → transform suspended and subgoal created

```
transform MoveSquareLeft
  a is a square
  consider if
    goal is to change location
    best place is to the left
  body
    pick up square
    move left
    put down square
```

Activation Table

- Specifies conditions and actions
  - Like truth tables or production rules
- Defined by:
  - Condition block
  - Action block

    Actions are labeled:
    - T (true)
    - F (false)
    - * (don't care)

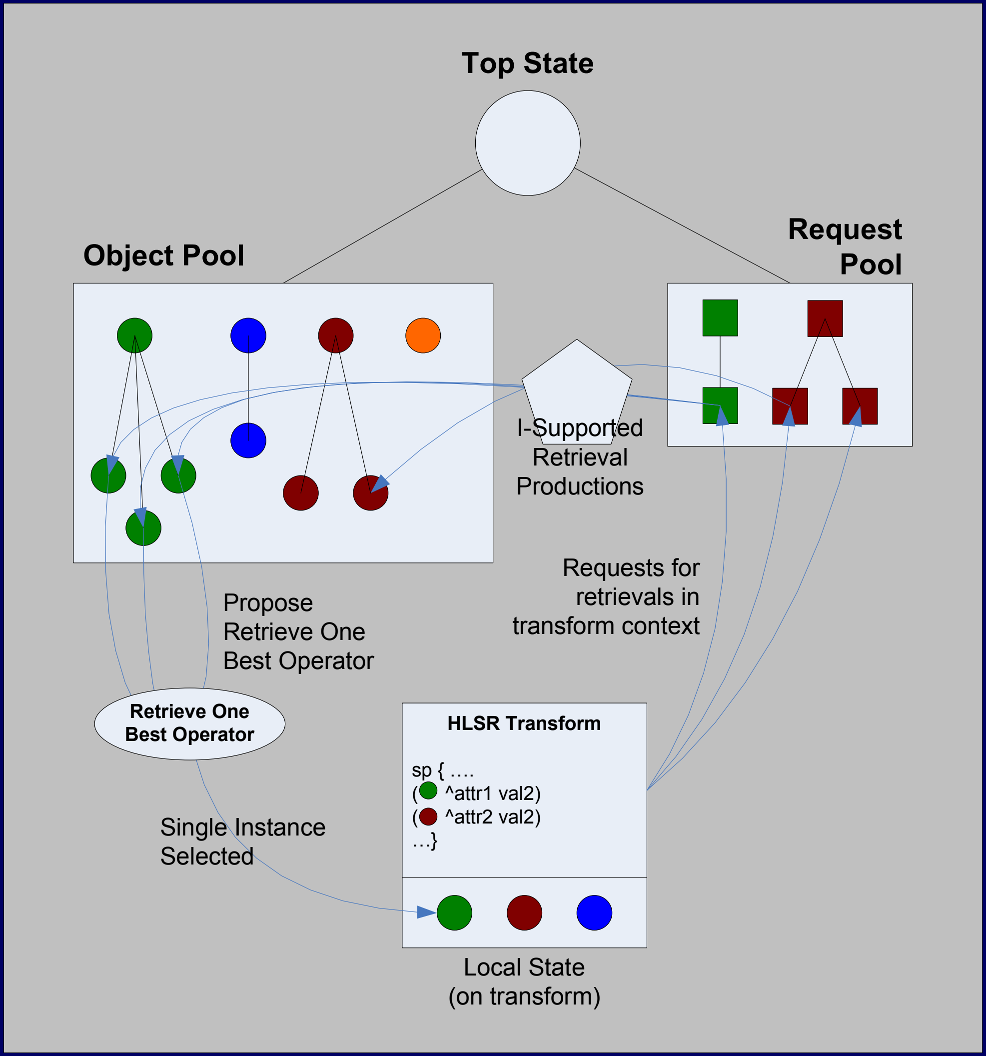
```
activation table WeatherGear
conditions
1: It is raining
2: It is windy

TT  Wear raincoat, no umbrella
TF  Wear raincoat, bring umbrella
F*   No raincoat, no umbrella
```

Compiling Relations to Soar

- **Key Requirements**
  - Blend asserted facts with computed facts (retrieve v. compute problem)
  - Map to HLSR global memory pool (no state references)
  - Retrieve one best (eliminate multiple retrievals)
- **Constraints**
  - Partial matches can cause significant slow down
- **Observation**: compiler lacks some of the semantic information humans use to do this efficiently
  - Cardinality constraints
  - Data lifetime: how long data is valid

A Soar Microtheory for Relations

- Objects placed in pools based on type
- Retrievals on demand
  - Transform assert requests in pool
  - I-supported productions assert value based on met condition
- Operator used to select one best
- Directly asserted and retrieved facts represented in object pool the same way



Top State

Object Pool

Request Pool

I-Supported Retrieval Productions

Requests for retrievals in transform context

Propose Retrieve One Best Operator

Retrieve One Best Operator

Single Instance Selected

HLSR Transform

```
sp { ….
(● ^attr1 val2)
(● ^attr2 val2)
…}
```

Local State (on transform)

Compiling Goals to Soar

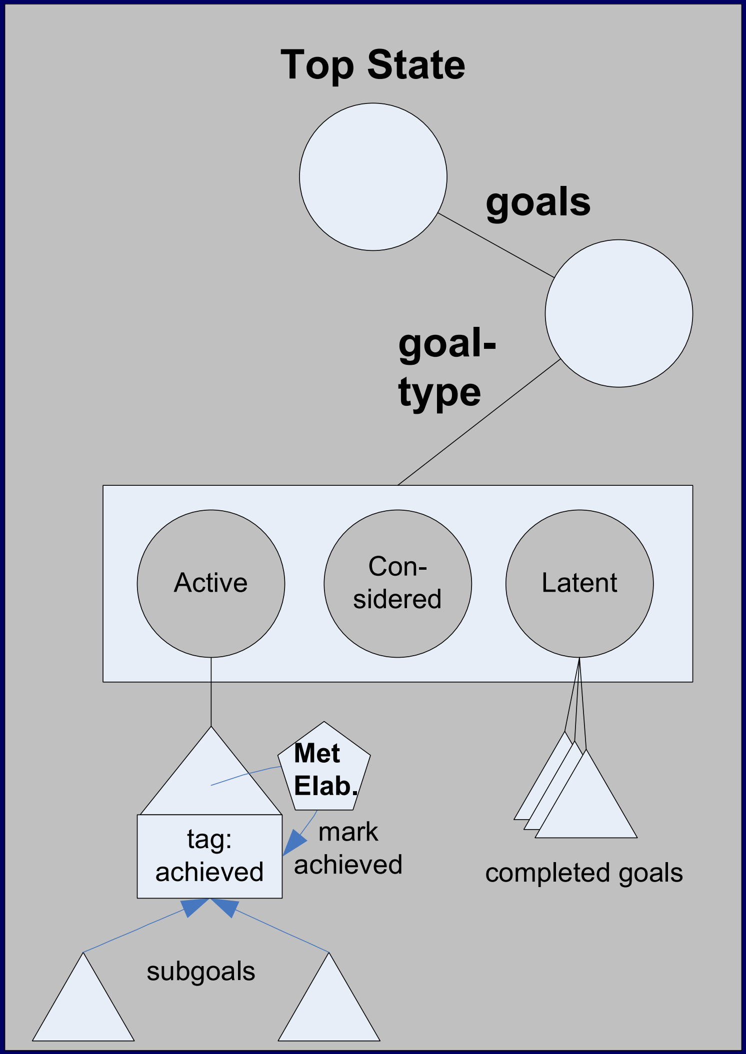
- **Key Requirements**
  - Represent goal forest
  - Auto-reconsideration via met condition
- **Constraints**
  - Soar "state stack" can only represent single thread of goals
- **Observation**: we have to decide how to leverage goal stack

  Our Current Approach

  - Use FOG approach similar to "Radical Randy" OR
  - Use FOG declarative representation but use state stack to have single thread of **active** goals

A Soar Microtheory for Goals

- Goals pooled in similar way to objects
  - Extra layer for active state of goal
  - Active goal pool should be small for performance
- "Met" condition elaborations mark goal achieved
- Operator used to move goal to the "Latent" bin after achievement

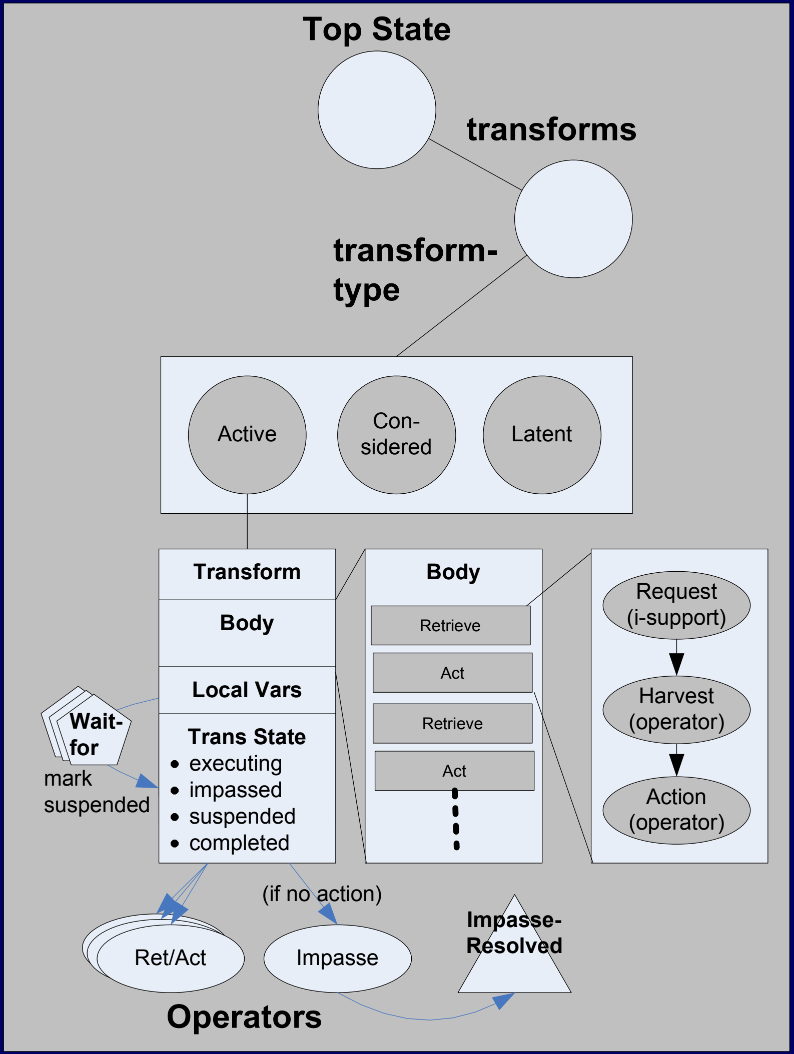


Compiling Transforms to Soar

- **Key Requirements**
  - Hold consistent variable bindings
  - Execute sequences including waitfor statements
  - Provide an automatic subgoal mechanism for transform failures
- **Constraints**
  - Soar is inherently parallel
- **Observation**: with transforms the compiler does most bookkeeping that developers usually do
  - Process tags and temporary variables
  - Sequence tags and conditions

A Soar Microtheory for Goals

- Transform objects store state and tags for multiple operators
  - Could be a Soar state
- Code generation decomposes body to retrieve/act pairs
- Each retrieve/act pair executed sequentially with tags used to control sequence
- Waitfors using i-support
- Impasses generated when no operator (i.e. state no change)



Compiling ACT-R

- Challenges
  - ACT-R much more sequential: few, narrowly defined points of parallelism
  - ACT-R has no support for predicate logic
  - ACT-R can be non-deterministic: what is the acceptable number of times it should get the “right? answer?
- ACT-R microtheories
  - Map complex retrievals to low level retrieval sequences
  - Leverage the goal (or context) buffer to represent processing state for all HLSR constructs
  - Provide less parallelism: generally the ACT-R program has to decide explicitly *when* to check conditions (e.g. met conditions, activation table conditions, etc)

Conclusions

- Status of the HLSR project
  - Initial implementation nearly complete
  - Evaluation on the way
  - Building abstractions and micro-theories has revealed interesting and subtle differences between architectures
- HLSR will:
  - **Abstract** away from details of a particular cognitive architecture
  - **Encapsulate** knowledge and behaviors
  - Improve **efficiency** of creating new models
  - Allow easier **comparisons** of models and architectures
  - Make cognitive modeling more accessible