ACT-R as Embedded Code

Dario Salvucci
Drexel University
Specifying ACT-R Models

- At this point, we have several implementations of the ACT-R architecture
  - LISP ACT-R
  - jACT-R
  - Java ACT-R
  - Python ACT-R
  - [Distract-R]
  - [ACT-RN]
  - [Javascript ACT-R]

- Different implementations of ACT-R have taken different approaches to specifying models…
Specifying ACT-R Models

- **Approach #1: Interpreted language**
  - #1(a): Canonical ACT-R
  - LISP ACT-R, Java ACT-R [+ Javascript ACT-R]
  - Same model regardless of the language of the underlying interpreter
  - (Choice of language has greater effect on task implementation)

```lisp
(sgp :esc t :lf .05)

(add-dm
  (a isa count-order first 0 second 1)
  ...
  (goal isa add arg1 5 arg2 2))
)

(P initialize-addition
  =goal>
  isa          add
  ...
  ==>
  =goal>
  sum          =num1
  count        0
  +retrieval>
  isa          count-order
  first        =num1
)
```
Specifying ACT-R Models

- **Approach #1: Interpreted language**
  - #1(b): Different language
  - e.g., jACT-R
  - Still, language of the underlying interpreter doesn’t matter

```xml
<chunk name="j" type="count-order">
  <slot name="first" equals="9.0"/>
  <slot name="second" equals="10.0"/>
</chunk>

<production name="initialize-addition">
  <conditions>
    <match buffer="goal" type="add">
      <slot name="arg1" equals="=num1"/>
      <slot name="arg2" equals="=num2"/>
      <slot name="sum" equals="nil"/>
    </match>
    <query buffer="retrieval">
      <slot name="state" equals="free"/>
    </query>
  </conditions>
  <actions>
    ...
  </actions>
</production>
```
Specifying ACT-R Models

- Approach #2: Embedded code

```python
class MyAgent(ACTR):
    focus=Buffer()

    DMbuffer=Buffer()
    DM=Memory(DMbuffer, latency=1.0, threshold=1)

    dm_n=DMNoise(DM, noise=0.0, baseNoise=0.0)
    dm_bl=DMBaseLevel(DM, decay=0.5, limit=None)

    def init():
        DM.add('customer:customer1 condiment:mustard')
        focus.set('rehearse')

    def request_chunk(focus='rehearse'):
        print "recalling the order"
        DM.request('customer:customer1 condiment:?condiment')
        focus.set('recall')
```
Specifying ACT-R Models

- Approach #2: Embedded code

```java
// control-attend-near
if ((na == NILVAL) && (when == NILVAL)
    && model.getVision().isVisionFree()
    && model.getVision().getVisualLocation() == null
    && model.getVision().getVisual() == null) {
    model.trace("DRIVE", "control-attend-near");
    na = NONEVAL;
    model.getVision().startVisualLocation(Chunk.KIND_NEAR);
    return true;
}

// control-attend-near-wait
if ((na == NILVAL) && (when != NILVAL) && (when <= model.getTime())
    && model.getVision().isVisionFree()
    && model.getVision().getVisualLocation() == null
    && model.getVision().getVisual() == null) {
    model.trace("DRIVE", "control-attend-near-wait");
    na = NONEVAL;
    model.getVision().startVisualLocation(Chunk.KIND_NEAR);
    return true;
}
```

Distract-R
Specifying ACT-R Models

- So, which approach is best?
- Each approach apparently has its value — why else would people have made them :)
- But it’s useful to think about some issues...
Issue #1: Constraints

- A modeling language constrains the user to specify knowledge/behavior in a very particular way
  - one of the hallmarks of a cognitive architecture
  - a unified approach to knowledge representation
- But often, rules are bent/broken to address components outside the model’s scope
  - e.g., the dreaded `eval`
  - not necessarily a bad thing
    - might just be a way to abstract over things beyond the model
  - if you’re concerned about building useful models, it can be used in a productive way
Issue #1: Constraints

- **Two examples...**
  - **Chunks vs. equations**
    - e.g., the driver model uses an equation to compute steering angle from visual points
      - this is really a stand-in for retrieval of chunks, learned over time... but abstracts over this issue for simplicity
  - **Dynamic chunks**
    - e.g., large-scale database of declarative chunks likely needs to be implemented differently
      - perhaps, create chunks on the fly, rather than storing all
      - (analogous problem to equations)
    - e.g., natural language
      - what if we wanted to store parts of speech in ACT-R? how might this be implemented?
Issue #2: Procedural Learning

- For an interpreted model, rules are created at the start, but can be changed on the fly
  - a la production compilation

- Embedded code doesn’t (easily) allow for procedural learning

- Embedded code also encourages a sequential style of behavior description — not as rules evaluated in parallel
  - in my mind, it seems to be an open question of how many models gain from this flexibility
Issue #3: Model Integration

- In theory, a modeling language facilitates integration of 2+ models
  - they’re all written in the same language, using the same cognitive representations

- In practice, as we know, this doesn’t happen much
  - the “API” between models is difficult to validate
  - embedded code helps to enforce the API
    - because of type checking, including packages/libraries, etc.
    - e.g., by defining types and specific slots

- Again, it comes down to what’s easy & useful
- Which brings us to our user base…
Potential Users

- We’ve largely targeted ACT-R to other cognitive scientists
  - they are trying to understand cognition
  - they care, first and foremost, about the model
  - user base: maybe 100-1000 people

- Meanwhile, there are plenty of “agent builders” interested in coding behavioral models
  - e.g., “behavior trees” for gaming
  - the “model” isn’t the 1st, or 7th, thing on their mind
  - they need something that integrates quickly and easily
  - potential user base: >>1000 people
“Agent Builder” Needs

- Get up and running quickly?
  - download a library, get code from a tutorial, integrate
  - interpreted ACT-R?
    - right now, fairly difficult, especially the glue between the task and what the model sees
  - embedded code?
    - potentially much faster — if the programming language matches

- Language interoperability?
  - they can’t conform to our language (they already have 100k lines of code in another language)
  - (we might spend lots of time integrating a Unity game with LISP code, but I doubt anyone else would)
“Agent Builder” Needs

- Access behavior at different levels of abstraction?
  - do they need an actual running model?
  - or are they looking for smaller functions??
    - e.g., calculate mouse movement or keystroke time
    - e.g., calculate response time for a visual search

- Visual editors and IDEs?
  - game behavior-tree designers rely heavily on these…
class ClickWorld extends World {
    private Display display;
    private Item button;

    ClickWorld() {
        super();
        display = new Display();
        button = new Item("button", 0, 0, 30, 30);
        button.addClickListener(new ClickListener() {
            @Override
            public void click() {
                moveButton();
            }
        });
        display.add(button, "X");
        moveButton();
    }

    void moveButton() {
        display.move(button, 50 + random.nextInt(200), 50 + random.nextInt(200));
        log("move");
    }

    public static void main(String args[]) {
        ClickWorld world = new ClickWorld();
        new Simulation(new ClickAgent(world)).setRealTime(true).run();
    }
}
Prototype System

class ClickAgent extends Agent {
    private ClickWorld world;
    private DesktopVision vision;
    private DesktopMotor motor;

    ClickAgent(ClickWorld world) {
        super();
        this.world = world;
        Display display = world.getDisplay();
        vision = new DesktopVision(this, display);
        motor = new DesktopMotor(this, display, vision);
    }

    @Override
    public void run() {
        while (!world.isDone()) {
            vision.waitFor(new Pattern(Item.TYPE, "button"));
            motor.pointAndClick(vision.getFound());
        }
    }
}
public class Numbers implements MemoryModule {
    ...

    public Numbers() { ... }

    public NumberChunk get(int n) { ... }

    @Override
    public void addStaticChunks(Memory memory) {
    }

    @Override
    public MemoryChunk getDynamicChunk(Pattern pattern) {
        NumberChunk chunk = null;
        if (pattern.has("isa", Operator.EQ, "number")) {
            SlotPattern slotPattern = pattern.get("value", Operator.EQ);
            if (slotPattern != null) {
                Integer value = (Integer) slotPattern.getValue();
                if (value != null)
                    chunk = get(value);
            }
        }
        return chunk;
    }
}
public class Counting extends Module {
    private Memory memory;
    private Speech speech;

    public Counting(Agent agent,
                    Memory memory, Numbers numbers, Speech speech) {
        super("counting", agent);
        this.memory = memory;
        this.speech = speech;
        memory.include(numbers);
    }

    public void count(int from, int to) {
        while (from <= to) {
            memory.recall(new Pattern("isa", "number")
                             .add(NumberChunk.VALUE, from));
            speech.say(memory.getRecalled().getString("name"));
            from = memory.getRecalled().getInteger("next");
        }
    }
}
Thinking Ahead

- The prototype system is still built for simulating and acting, not other levels of abstraction
- Let's look at some examples…
  - Visual Search
  - Arithmetic
  - List Memory
Thinking Ahead

- **Visual Search**
  - example: iLab Vision C++ Toolkit (Itti et al., USC)
  - takes raw image as input, can generate as output...

  - strong predictions, easy to integrate with other models
Thinking Ahead

- Arithmetic (Lebiere, 1998)
  - RT, small & large problems over time…

![Power Law Speedup Over Time](image)
**Thinking Ahead**

- **Arithmetic (Lebiere, 1998)**
  - % correct vs. incorrect responses (age 4)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>other</th>
</tr>
</thead>
<tbody>
<tr>
<td>1+1</td>
<td></td>
<td></td>
<td></td>
<td>86</td>
<td></td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1+2</td>
<td></td>
<td></td>
<td></td>
<td>70</td>
<td></td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>1+3</td>
<td></td>
<td></td>
<td></td>
<td>71</td>
<td></td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1+4</td>
<td></td>
<td></td>
<td></td>
<td>61</td>
<td></td>
<td>1</td>
<td>7</td>
<td>9</td>
<td>7</td>
<td>2</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1+5</td>
<td></td>
<td></td>
<td></td>
<td>50</td>
<td></td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2+1</td>
<td></td>
<td></td>
<td>79</td>
<td></td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>2+2</td>
<td></td>
<td></td>
<td>80</td>
<td></td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2+3</td>
<td></td>
<td></td>
<td>34</td>
<td></td>
<td>9</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2+4</td>
<td></td>
<td></td>
<td>29</td>
<td></td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>2+5</td>
<td></td>
<td></td>
<td>13</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3+1</td>
<td></td>
<td></td>
<td>79</td>
<td></td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3+2</td>
<td></td>
<td></td>
<td>55</td>
<td></td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3+3</td>
<td></td>
<td></td>
<td>48</td>
<td></td>
<td>9</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3+4</td>
<td></td>
<td></td>
<td>29</td>
<td></td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>3+5</td>
<td></td>
<td></td>
<td>18</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4+1</td>
<td></td>
<td></td>
<td>68</td>
<td></td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4+2</td>
<td></td>
<td></td>
<td>36</td>
<td></td>
<td>13</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>4+3</td>
<td></td>
<td></td>
<td>38</td>
<td></td>
<td>9</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4+4</td>
<td></td>
<td></td>
<td>34</td>
<td></td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4+5</td>
<td></td>
<td></td>
<td>18</td>
<td></td>
<td>11</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5+1</td>
<td></td>
<td></td>
<td>71</td>
<td></td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5+2</td>
<td></td>
<td></td>
<td>22</td>
<td></td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5+3</td>
<td></td>
<td></td>
<td>23</td>
<td></td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5+4</td>
<td></td>
<td></td>
<td>16</td>
<td></td>
<td>11</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5+5</td>
<td></td>
<td></td>
<td>34</td>
<td></td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
Thinking Ahead

- Arithmetic (Lebiere, 1998)
  - as a library? runnable actions, simpler functions...

```java
public class Arithmetic {

    public Arithmetic(int age) { ... }

    public int add(int x, int y) {
        // performs addition with RT, correctness
    }

    public double getProbabilityCorrect(int x, int y) {
        // returns probability correct for this age
    }

    public double getResponseTime(int x, int y) {
        // returns predicted RT for this age, with noise
    }

    ...
}
```
Thinking Ahead

- **List Memory** (e.g., Anderson, Bothell, Lebiere, Matessa, 1998)
  - as a library? ...

```java
public class ListMemory {

    public ListMemory() { ... }

    public void clear() { ... }

    public void add(String word) { ... } // assumes running time
    public void add(String word, double t) { ... } // specifies time

    public List<Recalled> recall(double t) {
        // given current time, returns recalled list with RT and errors
    }

    ...
}
```
Summary

- **Hunch:** Embedded code will facilitate integration and sharing — at least for “agent builders” — in a way that interpreted language doesn’t
  - at least, for domains where production learning isn’t necessary or critical

- Only sketches of a prototype system at this point

- The proof would come in implementations of sample domains, like arithmetic, list memory, etc.

- … and their use by actual users