

# Soar-RL Tutorial

Soar Workshop 31 – Nate Derbinsky

While waiting...

1. Make sure you have internet access

2. Download Soar 9.3.1

[soar.googlecode.com](http://soar.googlecode.com)

3. Download Graphviz

[www.graphviz.org](http://www.graphviz.org)

4. Download tutorial support files

[www.eecs.umich.edu/~nlderbin/workshop31](http://www.eecs.umich.edu/~nlderbin/workshop31)

# Setting Expectations

This is **not** a tutorial on RL

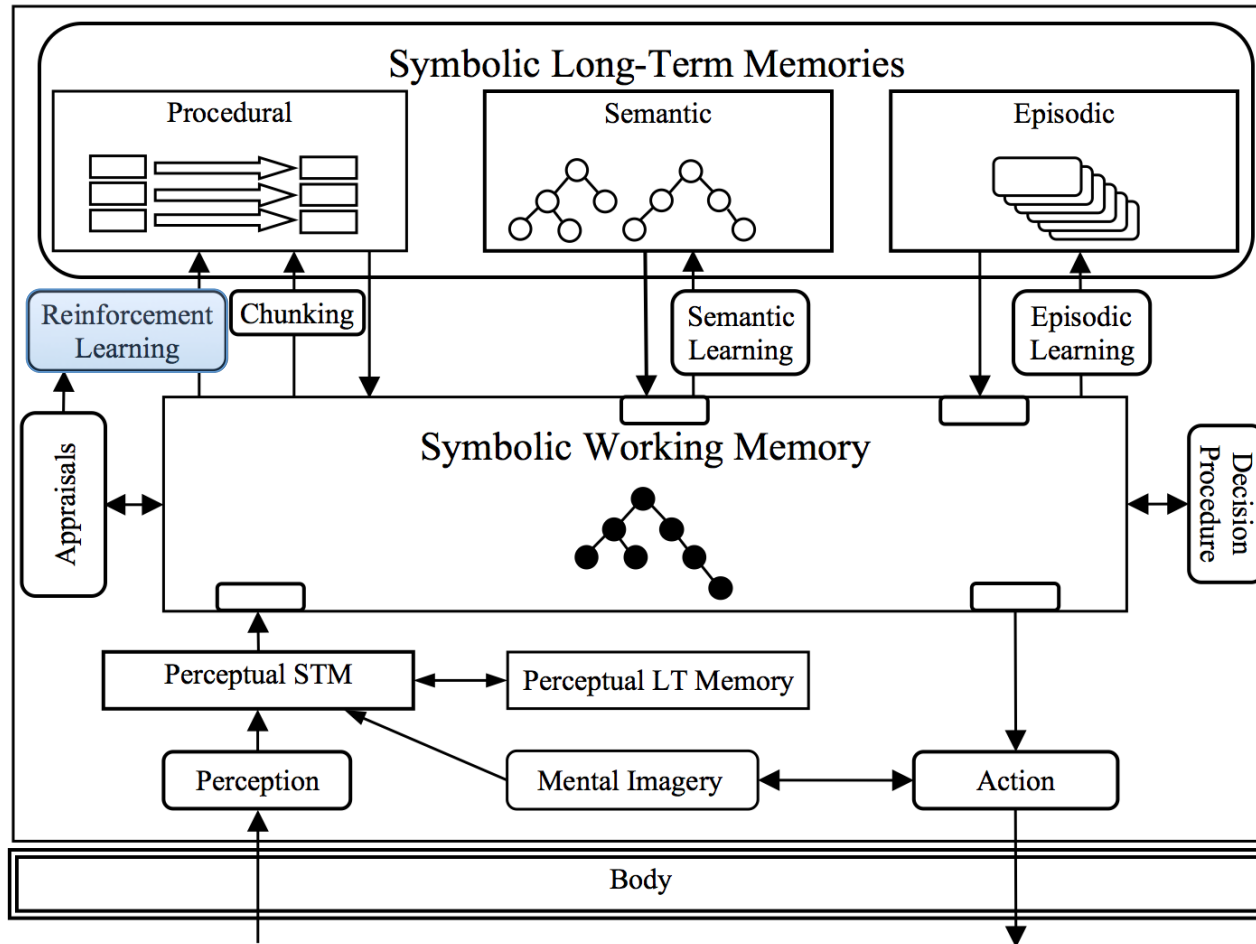
Reinforcement Learning: An Introduction

*Richard S. Sutton, Andrew G. Barto*

## Topics

- Soar-RL as a learning mechanism
- Agent design
- Advanced issues
- Additional resources

# Soar 9



# Procedural Learning Mechanisms

## Chunking

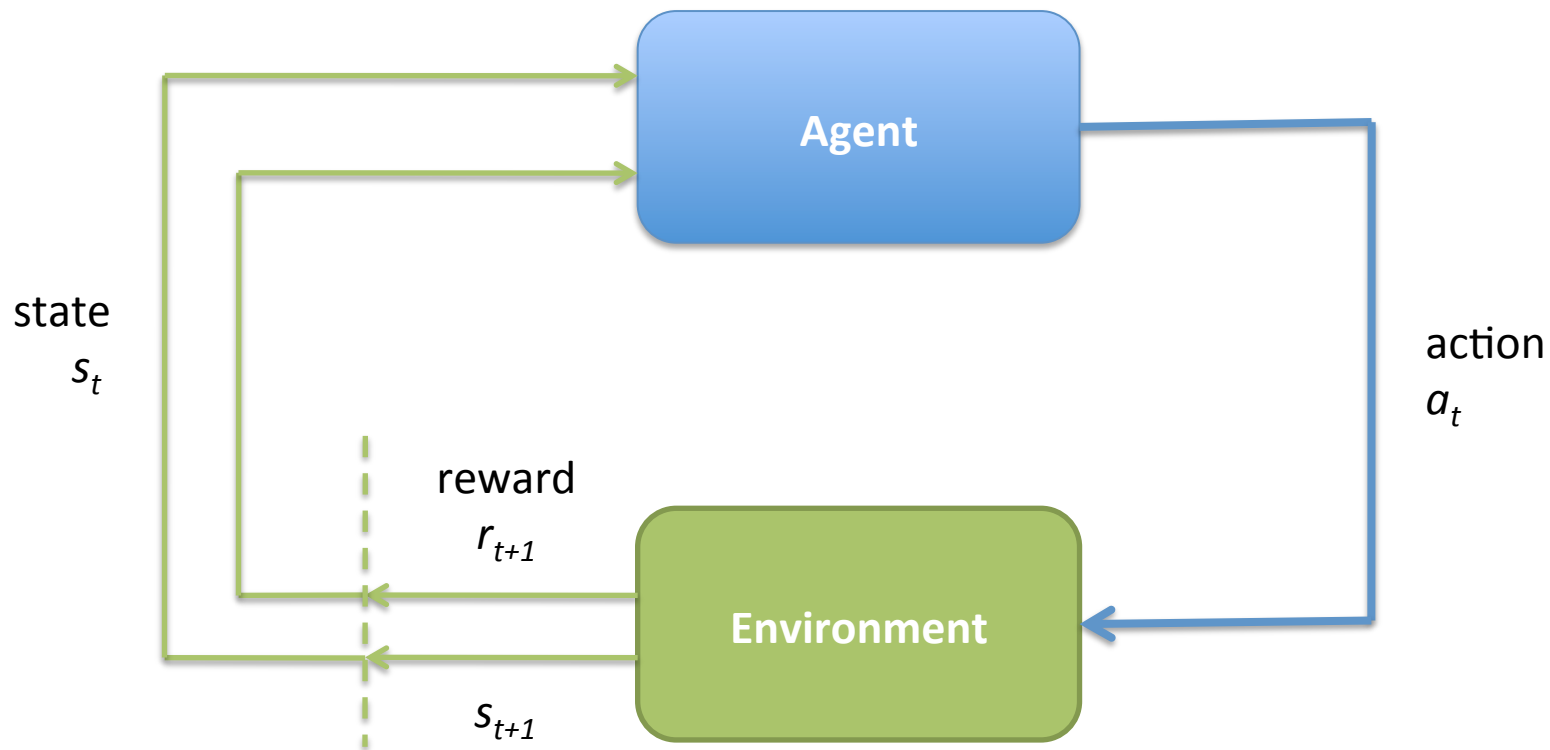
- Converts *deliberation* in substates into *reaction* via rule compilation
- Creates new rules

## Reinforcement Learning

- *Tunes* operator numeric preferences to reflect expectation of reward
- Updates existing rules

# Reinforcement Learning Cycle

Goal: action selection policy such as to maximize expected receipt of future reward



# Left-Right Demo

1. Soar Java Debugger
2. Source left-right agent

`share/soar/Demos/left-right.soar`



# Left-Right Demo

## *Script*

1. `srand 5041231`
2. `step`
3. `run 1 -p`
4. `click: op_pref tab`
  - note numeric indifferents
5. `print left-right*rl*left`
6. `print left-right*rl*right`
7. `run`
  - note movement direction
8. `print left-right*rl*left`
9. `print left-right*rl*right`
10. `init-soar`
11. Repeat from #2 (~5 times)

# Left-Right: Takeaways

Reinforcement learning changes rules in procedural memory

- Changes are persistent
- Change affects numeric indifferent preferences, which in turn affects the selection of operators
- Change is in the direction of the underlying reward signal (will discuss this more shortly)



# Components of Soar-RL

- RL rules
- Reward representation
- Learning

# RL Rules

The RL mechanism maintains Q-values for state-operator pairs in specially formulated rules, identified by syntax

- RHS must be a single action, asserting a single numeric indifferent preference with a constant value

```
sp {left-right*rl*left
  (state <s> ^name left-right
    ^operator <op> +)
  (<op> ^name move
    ^dir left)
-->
  (<s> ^operator <op> = 0)
}
```

```
sp {left-right*rl*right
  (state <s> ^name left-right
    ^operator <op> +)
  (<op> ^name move
    ^dir right)
-->
  (<s> ^operator <op> = 0)
}
```

# Pop Quiz!

Which of the following are valid RL rules?

✘ sp {rule1  
 (state <s> ^name quiz  
 ^operator <op> +)  
 (<op> ^name op-name)  
-->  
 (<s> ^operator <op> = 1, >)}  
}

✘ sp {rule2  
 (state <s> ^name quiz  
 ^operator <op> +)  
 (<op> ^name op-name)  
-->  
 (<s> ^operator <op> = 0.8)  
 (write |debugging|)}  
}

😊 sp {rule3  
 (state <s> ^name left-right  
 ^operator <op> +)  
 (<op> ^name move  
 ^dir right)  
-->  
 (<s> ^operator <op> = -1.4)}  
}

✘ sp {rule4  
 (state <s> ^name left-right  
 ^operator <op> +)  
 (<op> ^name move  
 ^dir right)  
-->  
 (<s> ^operator <op> = <one>)}  
}

# Left-Right Demo

*Focus: RL Rules*

1. Soar Java Debugger
2. Source left-right agent  
`share/soar/Demos/left-right.soar`
3. `print --full --rl`
4. `run`
5. `print --full --rl`
6. `print --rl`
7. Attempt sourcing rules from previous slide
  - Check with `print --rl`

# Reward Representation

Soar creates a `reward-link` structure on each state in WM

- Soar Java Debugger

1. `step 5`

2. `print --exact (* ^reward-link *)`

Reward is recognized by syntax

```
(<reward-link> ^reward <r>)
```

```
(<r> ^value [val])
```

- [val] must be a numeric constant (integer or float)
- The reward-link is **not** directly modified by IO
- The reward-link is **not** automatically cleaned
- Reward is collected at the beginning of each *decide* phase
- Reward on a state's reward-link pertains to that state (more on this later)
- Multiple reward values are summed by default

# Reward Rule Examples

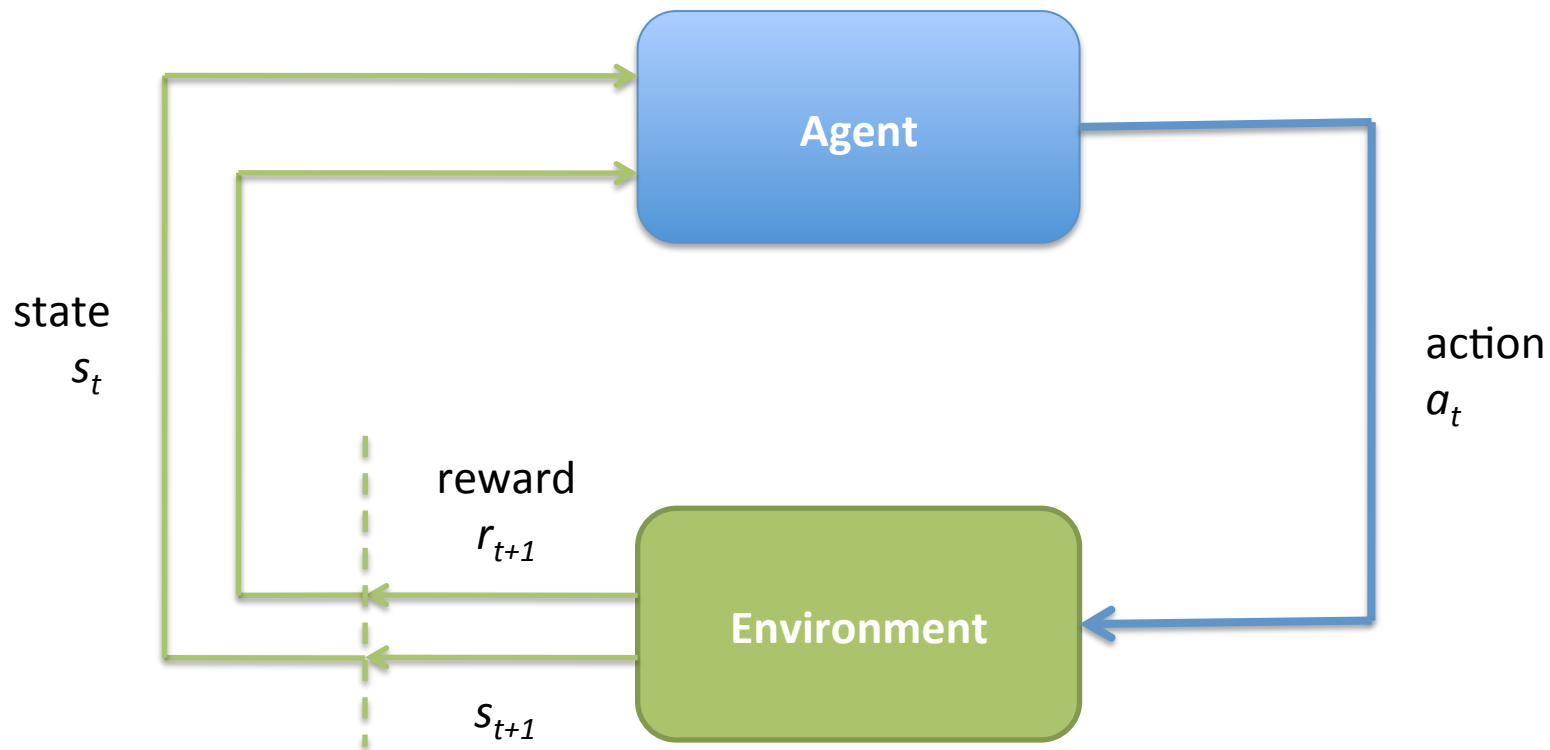
```
sp {left-right*reward*left
    (state <s> ^name left-right
      ^location left
      ^reward-link <rl>))
-->
    (<rl> ^reward <r>)
    (<r> ^value -1)
}
```

```
sp {left-right*reward*right
    (state <s> ^name left-right
      ^location right
      ^reward-link <rl>))
-->
    (<rl> ^reward <r>)
    (<r> ^value 1)
}
```



# Reinforcement Learning Cycle

Goal: action selection policy such as to maximize expected receipt of future reward



# Soar-RL Cycle

Decision	Input	Propose	Decide	Apply	Output
d					
d+1					



# Soar-RL Cycle

Decision	Input	Propose	Decide	Apply	Output
<b>d</b>	state <sub>d</sub>				
<b>d+1</b>					

# Soar-RL Cycle

Decision	Input	Propose	Decide	Apply	Output
<b>d</b>	state <sub>d</sub>	evaluate operators <sub>d</sub>			
<b>d+1</b>					

# Soar-RL Cycle

Decision	Input	Propose	Decide	Apply	Output
<b>d</b>	state <sub>d</sub>	evaluate operators <sub>d</sub>	select operator <sub>d</sub>		
<b>d+1</b>					

# Soar-RL Cycle

Decision	Input	Propose	Decide	Apply	Output
<b>d</b>	state <sub>d</sub>	evaluate operators <sub>d</sub>	select operator <sub>d</sub>		initiate external action(s)
<b>d+1</b>					

# Soar-RL Cycle

Decision	Input	Propose	Decide	Apply	Output
<b>d</b>	$state_d$	evaluate operators <sub>d</sub>	select operator <sub>d</sub>		initiate external action(s)
<b>d+1</b>	$state_{d+1}$ reward <sub>d+1</sub>				

# Soar-RL Cycle

Decision	Input	Propose	Decide	Apply	Output
<b>d</b>	$state_d$	evaluate operators <sub>d</sub>	select operator <sub>d</sub>		initiate external action(s)
<b>d+1</b>	$state_{d+1}$ $reward_{d+1}$	evaluate operators <sub>d+1</sub>			

# Soar-RL Cycle

Decision	Input	Propose	Decide	Apply	Output
<b>d</b>	state <sub>d</sub>	evaluate operators <sub>d</sub>	select operator <sub>d</sub>		initiate external action(s)
<b>d+1</b>	state <sub>d+1</sub> reward <sub>d+1</sub>	evaluate operators <sub>d+1</sub>	select operator <sub>d+1</sub> update policy <sub>d</sub>		

# RL Updates

- Takes place during *decide* phase, after operator selection
- For all RL rule instantiations ( $n$ ) that supported the *last* selected operator

$$\text{value}_{d+1} = \text{value}_d + (\delta_d/n)$$

Where, roughly...

$$\delta_d = \alpha[\text{reward}_{d+1} + \gamma(q_{d+1}) - \text{value}_d]$$

Where...

- $\alpha$  is a parameter (learning rate)
- $\gamma$  is a parameter (discount rate)
- $q_{d+1}$  is dictated by learning policy
  - On-policy (SARSA): value of selected operator
  - Off-policy (Q-learning): value of operator with maximum selection probability



# Left-Right Demo

*Focus: RL Updates*

1. Soar Java Debugger

2. Source left-right agent

```
share/soar/Demos/left-right.soar
```

3. `watch --rl`

4. `run`

5. `print s1`

6. `print --depth 2 rl`

7. `rl --stats`

$$\delta = 0.3[1 + 0.9(0) - 0] = 0.3$$

8. `rl`

$$\delta = 0.3[1 + 0.9(0) - 0.3] = 0.21$$

# Controlling Soar-RL

Get/Set a parameter:

- `rl [-g|--get] <name>`
- `rl [-s|--set] <name> <value>`

Try changing learning-rate to 1 (deterministic)

1. `rl`
2. `rl --set learning-rate 1`
3. `rl`

# RL Agent Design

1. RL rules
2. Reward rules
3. Enable RL

Baseline: `left-right-start.soar`

[www.eecs.umich.edu/~nlderbin/workshop31/rl](http://www.eecs.umich.edu/~nlderbin/workshop31/rl)

# 1. RL Rules

*left-right-1.soar*

- a) Add RL rules for moving left/right (slide 10)
  - Check via `print --rl`
  
- b) Remove indifferent preference from operator proposal rules
  - Note: this will cause a tie impasse if (a) is not done correctly

## 2. Reward Rules

*left-right-2.soar*

Add rules to add reward to reward-link,  
dependent upon the location to which the agent  
decided to move (slide 14)

# 3. Enable RL

*left-right-3.soar*

By default, Soar-RL is **disabled**. To enable, set the parameter (slide 26):

```
rl --set learning on
```

Note: the agent will function without this step, but rules will not update, and hence will act identically to the baseline agent.

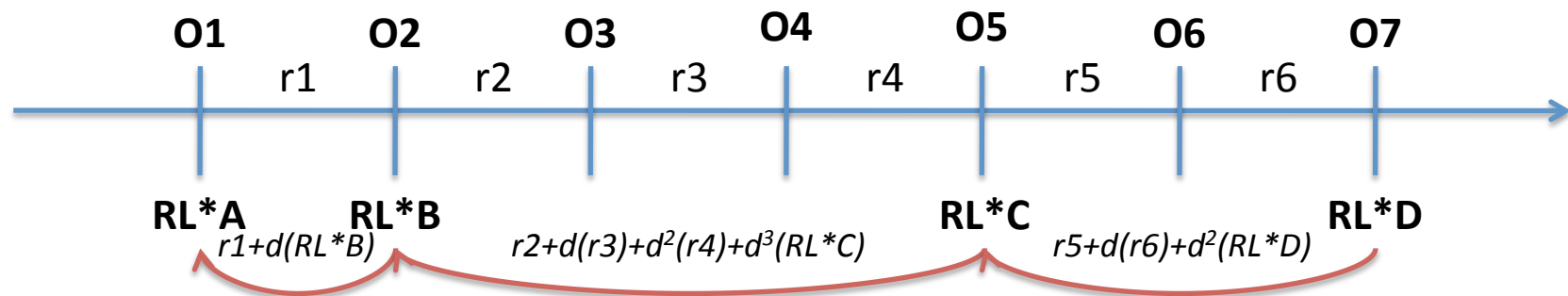
# Advanced Issues

- Gaps in rule coverage
- RL in substates
- Exploration policies
- Rule generation

# Gaps in Rule Coverage

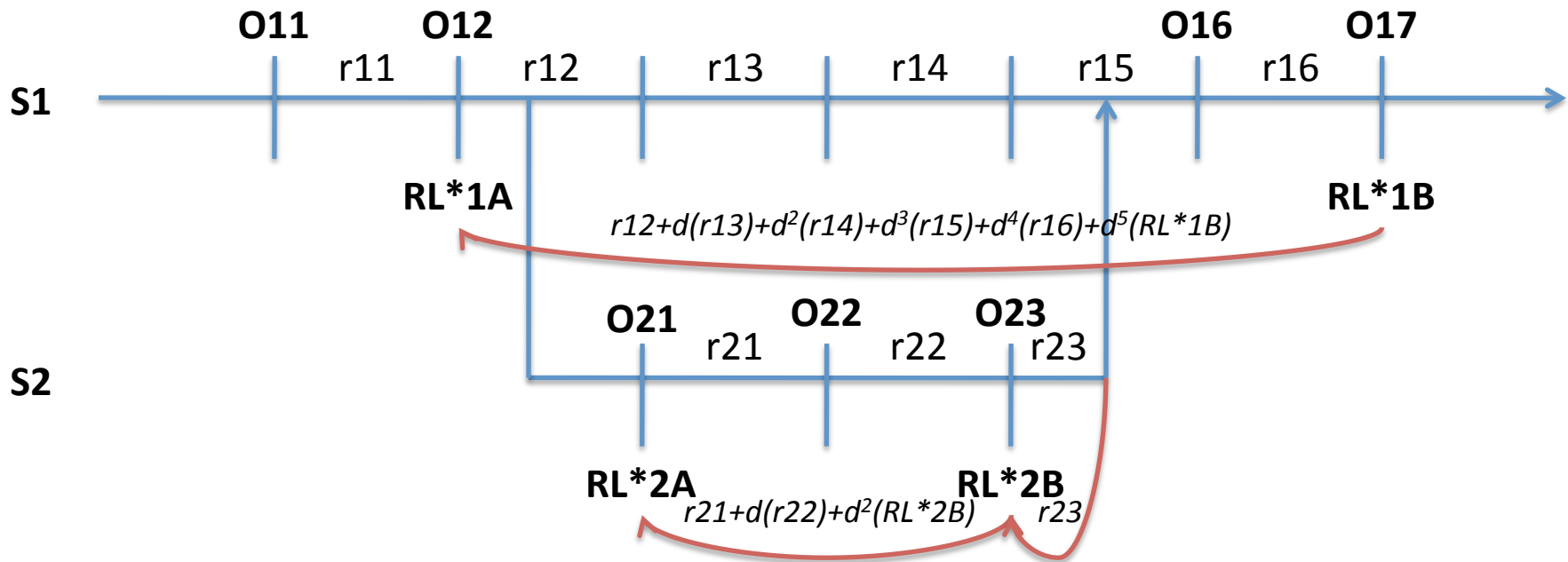
**Gap:** one or more contiguous decision cycles during which no RL rules fire

By default, Soar-RL will automatically propagate RL updates over gaps, where rewards are discounted with respect to the length of the gap





# RL in Substates



- Rewards are collected independently on each state
- Rewards at a superstate are attributed to the last RL-supported, selected operator

# Exploration Policies

There are numerous policies for selecting operators probabilistically (see Manual)

- Deterministic (first, last)
- Softmax (default)
- Epsilon Greedy (default w/ RL)
- Boltzmann

Pertinent commands (see Manual)

- `indifferent-selection`
- `predict`
- `select`

# Rule Generation

The number of RL rules required for an agent is usually unfeasibly large to write by hand

## Options

- gp command
- Rule templates
- Chunking

# gp Command

- Similar syntax to `sp` command
- Generates rules, at source time, according to the cross-product of a fixed number of dimensions, each with a fixed domain
  - Very fast, potentially creates a lot more rules than will be encountered by the agent

```
gp {example*gp                                > Total: 36 productions sourced.
  (state <s> ^name left-right                 > print
    ^operator <op> +
    ^d1 [ a b c ]
    ^d2 [ 1 2 3 ])
  (<op> ^name move
    ^dir [ left right up down ])
-->
  (<s> ^operator <op> = 0)
}
```

# Left-Right Demo

*Focus: gp (left-right-3-gp.soar)*

Replace RL rules (slide 28) with a single gp command

- Source agent
- Verify rules: `print --full --rl`

# Rule Templates

- Allows Soar to dynamically generate new RL rules based on a predefined pattern as the agent encounters novel states
- Similar to syntax of RL rules
  - Requires `:template` flag
    - Note: without will be an RL rule that matches multiple states
  - Numeric indifferent preference value can be a variable
- Behavior
  - Rather than firing, creates new RL rules from instantiations
  - Significantly slower than `gp`

# Left-Right Demo

*Focus: template (left-right-3-template.soar)*

Replace RL rules (slide 28) with a single rule template

1. Source agent

2. Verify rules

➤ `print --full --rl`

➤ `print --full --template`

3. run

4. Verify rules

➤ `print --full --rl`

➤ `print --full --template`

# Chunking

*“Since RL rules are regular productions, they can be learned by chunking just like any other production. This method is more general than using the gp command or rule templates, and is useful if the environment state consists of arbitrarily complex relational structures that cannot be enumerated.”*

- Soar Manual



# Additional Resources

- Documentation
- Demo agents
- Readings

# Documentation

## Manual and Tutorial

`share/soar/Documentation`

## Additional Topics

- RL update details
- Eligibility traces
- Additional learning agent walkthrough (Water-jug)
- Usage: commands, parameters, statistics, etc.
- ...

# Demo Agents

share/soar/Demos

- Left-Right
- Water Jug RL
  - Discussed in Soar-RL Tutorial
  - Accessible from Soar Java Debugger *Demos* menu
- RL Unit
  - Exemplifies update behavior over gaps/sub-goals

# Select Readings

*code.google.com/p/soar/wiki/Publications*

2005

- Soar-RL: Integrating Reinforcement Learning with Soar
  - Shelley Nason, John E. Laird (Cognitive Systems Research)

2007

- The Importance of Action History in Decision Making and Reinforcement Learning
  - Yongjia Wang, John E. Laird (ICCM)

2008

- A Computational Unification of Cognitive Control, Emotion, and Learning
  - Robert P. Marinier III (Dissertation)

2009

- Learning to Play Mario
  - Shiwali Mohan (Technical Report)
- Hierarchical Reinforcement Learning in the Taxicab Domain
  - Mitchell K. Bloch (Technical Report)

2010

- Instance-Based Online Learning of Deterministic Relational Action Models
  - Joseph Xu, John E. Laird (AAAI)
- Using Imagery to Simplify Perceptual Abstraction in Reinforcement Learning Agents
  - Samuel Wintermute (AAAI)

2011

- Learning to Use Episodic memory
  - Nicholas Gorski, John E. Laird (Cognitive Systems Research)

# Extra Time

- Extend Water-Jug base agent to use RL
  - See Soar-RL Tutorial for guidance