Soar-RL Manual

Version 1.0.1
March 25, 2010

Contributors
Nate Derbinsky
Nick Gorski
John Laird
Bob Marinier
Yongjia Wang

1. Table of Contents

2. Document History

Version 1.0.1
Added HRL Discount parameter.

Version 1.0
Soar 9.0 release.

Version 0.1
Initial specification.

3. Soar-RL Motivation

Soar-RL is the architectural integration of reinforcement learning (RL) with Soar. The RL mechanism will automatically learn value functions as a Soar agent executes. These value functions represent, for a given working memory state and proposed operator, the expected sum of future rewards the agent will receive if it selects that operator.

Optimal behavior for an agent is defined by reward and a discount factor. The agent acts so as to maximize the expected value:

$$r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots$$

where r_{t+1}, r_{t+2}, r_{t+3}, … are the rewards at future time steps, and γ is the discount factor.

4. Working Memory Structure

Upon creation of a new state within working memory, the architecture will automatically create a structure in working memory called **reward-link**. The **reward-link** contains working memory elements populated by the agent to represent reward values (see *5. Reward* for more details).

5. Reward

A Soar-RL programmer must provide reward information for the agent to properly learn behavior.

5.1. Reward Location

Rewards are numeric values in the range (-∞,∞). Soar-RL gets reward from a state's **reward-link**. The value function for an operator at a particular level in the state stack is affected only by rewards at that level.

The location for a reward is a numeric element on the **reward** attribute of the **reward-link** (i.e. state.reward-link.reward.*). The element should be either an integer or float. For instance, the following reward attributions are correct:

```
state ^reward-link.reward.value 1.2

state ^reward-link.reward.value -2
```

All non-numeric elements at this level, or any values above/below this level, will be ignored.

5.2. Environmental Reward

The **reward-link** is not part of the **io-link** and is not modified directly by the environment. Reward information from the environment should be copied from the **input-link** to the **reward-link**.

5.3. Accumulation of Reward

Rewards will be collected (and summed if there are multiple rewards) at the beginning of each decision phase. Rewards are not removed from working memory as they are recorded, so, for instance, an o-supported reward will continue to be counted until it is explicitly removed.

As special cases, reward accumulation will occur immediately after a **halt** and immediately before sub-state retraction (see *7.5. Hierarchical Learning*).

6. Soar-RL Rules

The reinforcement learning algorithm aggregates the agent's experience into the Q-function. This function, **Q(s,a)**, is a mapping from a state-operator pair (described by a working memory state and a reference to a particular proposed operator) to a real number (the Q-value). The Q-value represents the expected discounted sum of future rewards if the agent selects the given operator and continues to follow its current policy. RL learns successively closer approximations to the true Q-values. Soar-RL stores this value function in productions so that the Q-value of an operator is computed from all rules that create numeric preferences for it during a decision.

6.1. Rule Format

The value function is computed through numeric preferences. Numeric preferences take the following form:

```
(<state variable> ^operator <operator variable> = number)
```

where **number** is a numeric constant.

The value function is stored in Soar-RL rules, which are Soar productions asserting a single numeric preference. Most simply:

```
sp {my*reinforcement*learning*rule
    (state <s> ^operator <o> +)
-->
    (<s> ^operator <o> = 2.3)
}
```

Soar-RL rules are identified by syntax. A rule is a Soar-RL rule if and only if its right hand side (RHS) consists of a single numeric preference (and it is not a template rule, described later). This format exists to ease technical requirements of identifying/updating Soar-RL rules, as well as to make it easy for the agent programmer to add/maintain RL capabilities within an agent.

Consider the following [non-Soar-RL] rules:

```
sp {multiple*preferences
    (state <s> ^operator <o> +)
-->
    (<s> ^operator <o> = 5, >)
}

sp {variable*binding
    (state <s> ^operator <o> +
               ^value <v>)
-->
    (<s> ^operator <o> = <v>)
}
```

The first rule proposes multiple preferences for the proposed operator and thus does not comply with the rule format. The second rule does not comply because it does not provide a *constant* for the numeric preference value.

6.2. Rule Behavior

In the simplest case, a single Soar-RL production fires and matches a particular operator. The estimated Q-value for the operator is the value of the numeric preference. This case corresponds to a tabular or state-aggregation representation of the Q-function.

In more complicated cases, multiple Soar-RL rules may fire for a single operator, proposing multiple numeric preferences for that operator. In this situation, the estimated Q-value for the operator is a function of the proposed numeric preferences. This function is controlled in Soar using the following command:

```
numeric-indifferent-mode [--sum|--avg]
```

where the **sum** option (default) sums over all the numeric preferences and **avg** averages them.

Soar-RL rules are Soar productions, with one unique property: Soar-RL rules are updated in production memory by Soar-RL. These updates, described later, change the **number** in the numeric preference, leaving the rest of the rule unaffected.

6.3. Template Format

Template rules have variables that are filled in to generate Soar-RL rules for state-operator pairs that the agent actually encounters. Consider the following template rule:

```
sp {sample*template*rule
    :template
    (state <s> ^operator <o> +
               ^value <v>)
-->
    (<s> ^operator <o> = 2.3)
}
```

The **:template** flag means to use the rule to make new Soar-RL rules by filling in those variables that match constants (**<v>** in this case) in working memory. Without the **:template** flag, this would have been a single Soar-RL rule that would match to multiple states.

A rule is a template rule if and only if it has the **:template** flag and, in all other respects, adheres to the format of a Soar-RL rule. However, wherein a Soar-RL rule may only use constants as the numerical preference value, a template rule may use a variable. Consider the following template rule:

```
sp {sample*template*rule2
    :template
    (state <s> ^operator <o> +
               ^value <v>)
-->
    (<s> ^operator <o> = <v>)
}
```

In this case, Soar-RL will create new productions whose numerical preference values are *initialized* to the value of **<v>** at the time this template rule first matches. If, at this time, the value of **<v>** is non-numeric, the numerical preference value is initialized to zero.

6.4. Template Behavior

Upon adding a production to production memory, Soar-RL checks for the existence of a **:template** flag. If a rule contains this flag and validates as a template rule, it is categorized as a template. If a rule contains this flag but is not a valid rule, it is immediately excised.

During the proposal phase, a valid template rule is supplied to the matcher as would any other rule. Matched instances of the rule, however, do not directly contribute preferences. Instead, they are used to create new Soar-RL productions. Each matched instance is compared against existing Soar-RL rules. If the LHS of the instance is not unique, the instance is ignored. If the LHS of the instance is unique, a new Soar-RL rule is added to production memory. It should be noted that the current process of identifying unique template match instances can become quite expensive in long agent runs. For performance reasons, if all Soar-RL productions can be predicted at agent design time, it is recommended to pre-generate them manually, using Soar's *gp* command, or via custom scripting.

A Soar-RL rule created from a template has two special characteristics: production name and constant replacement. The new production's name adheres to the following pattern: **rl*template-name*id**, where **id** is a unique identifier and **template-name** is the name of the originating template rule. The unique identifier is an incrementing counter maintained automatically for the agent. When a new production is generated, it receives an **id** value greater than the greatest identifier in production memory. The counter is updated during every successful instantiation of a Soar-RL rule from a template, as well as creation of productions (such as when sourcing an agent) that match the above naming scheme.

All variables in the new production that map to constants (as opposed to structures) are replaced with these constant values. This replacement applies to both the LHS conditions, as well as numerical preference value.

For example, consider **sample*template*rule2** above. Assume that the first time this template matches the value of **<v>** is **3.2**. The following new Soar-RL rule is added to production memory during the proposal phase:

```
sp {rl*sample*template*rule2*1
    (state <s> ^operator <o> +
               ^value 3.2)
-->
    (state ^operator <o> = 3.2)
}
```

As with other Soar-RL rules, the value of **3.2** on the RHS of this rule may be updated later by Soar-RL, whereas the value of **3.2** on the LHS will remain unchanged.

7. Reinforcement Learning Algorithm

The Soar-RL algorithm has the following major components: operator selection, preference updates, gaps in rule coverage, eligibility traces, and hierarchical learning.

7.1. Operator Selection

The purpose of learning a Q-function is that the agent can act optimally by selecting the operator with the highest Q-value. Numeric preferences participate in Soar's existing operator selection methods. The decision phase is further complicated by the exploration/exploitation needs of reinforcement learning.

7.1.1. Numeric and Symbolic Preferences

Symbolic preferences take precedence over numeric preferences. Symbolic preferences are processed first, and only if there are tied operators remaining, are numeric preferences examined. Consider the following example set of preferences:

```
O1 > O2
O1 = 0
O2 = 2.1
```

In this situation, **O1** would be selected.

7.1.2. Exploration Policies

When operator selection comes down to numeric preferences, the decision mechanism should usually choose the operator with the highest numeric preferences, that is, the highest estimated Q-value. However, for reinforcement learning to discover the optimal policy, it is necessary that the agent sometimes choose an action that does not have the maximum predicted value. Such exploration is necessary because actions may be undervalued. This situation can occur both during the initial learning of a task and as a result of change in the dynamics or reward structure of a task.

The exploration policy is selected using the **indifferent-selection** command:

```
indifferent-selection <policy>
```

The following are available policies:

-b, --boltzmann If the agent has proposed operators $\mathbf{O_1}$, ..., $\mathbf{O_n}$ with expected values $\mathbf{Q(s, O_1)}$, …, $\mathbf{Q(s, O_n)}$, then the probability of operator $\mathbf{O_i}$ being selected is

$$\frac{e^{Q(s,O_i)/\tau}}{\sum_{j=1}^{n} e^{Q(s,O_j)/\tau}}$$

	where τ (temperature), controls the peakedness of this probability distribution.
-g, --epsilon-greedy	With probability ε (epsilon) the agent selects an action at random (with uniform probability). Otherwise the agent takes the action with the highest expected value.
-x, --softmax	Select an operator at random from the set of mutually indifferent proposals, with the selection biased probabilistically by any existing numeric preferences. Preferences with non-positive numeric indifferent values are ignored. If non-positive numeric indifferent values are encountered, a purely random selection is made. Note: the softmax policy is analogous to the former "--random" option of the indifferent-selection command.
-f, --first	Deterministic. Select the first indifferent object from Soar's internal list.
-l, --last	Deterministic. Select the last indifferent object from Soar's internal list.

In an effort to maintain backwards compatibility, the default exploration policy is **softmax**. However, the first time that Soar-RL is enabled, the architecture changes this policy to **epsilon-greedy** (a more suitable default for RL agents) and issues a message to the trace.

Calling the **indifferent-selection** command with no parameters returns the current policy. For example, assuming defaults and Soar-RL is enabled:

```
>indifferent-selection
epsilon-greedy
```

If the exploration policy requires tempering, **indifferent-selection** uses an exploration rate parameter. Configuration of these parameters proceeds as follows:

```
indifferent-selection <parameter command> <value>
```

The following are available parameters:

Parameter Name	Parameter Command	Range of Values	Default Value
epsilon	**-e, --epsilon**	**[0,1]**	**0.1**
temperature	**-t, --temperature**	**(0, ∞)**	**25**

Calling the **indifferent-selection** command with the parameter name and no value returns the current parameter value. For example, assuming default values:

```
>indifferent-selection --epsilon
0.1
```

It should be noted that deliberate configuration of the **epsilon** parameter while using the **epsilon-greedy** exploration policy can achieve two extreme behaviors. If **epsilon** is set to a value of zero (0), there is no chance for exploration and the highest valued operator is always chosen (with random selection amongst tied operators). If **epsilon** is set to a value of one (1), a uniform random selection is always made from amongst the candidates.

With regard to reinforcement learning, the literature suggests that reduction of the exploration rate over time results in convergence of the Q-function to optimal. The **indifferent-selection** command allows for a reduction policy of each exploration rate parameter. The reduction policy (paired with a set of reduction rates) defines how the exploration rate parameter is reduced each cycle during which it is relevant (as defined by the currently selected **indifferent-selection** policy). Selection of reduction policy proceeds as follows:

```
indifferent-selection [-p|--reduction-policy]
      <parameter name> <reduction policy>
```

The <**parameter name**> comes from the parameter table above. The following are available reduction policies:

exponential	Exploration rate parameter decays exponentially (default).
linear	Exploration rate parameter decays linearly (value >= 0).

For example, setting the reduction policy for the **epsilon** parameter to **linear** would entail the following:

```
>indifferent-selection --reduction-policy epsilon linear
```

A call to the **reduction-policy** switch with a parameter name, but no reduction policy, will return the current reduction policy for the parameter. For example, after the command above:

```
>indifferent-selection --reduction-policy epsilon
linear
```

Configuring the reduction rate for a parameter in a particular reduction policy proceeds as follows:

```
indifferent-selection [-r|--reduction-rate]
      <parameter name> <reduction policy> <reduction rate>
```

The range and default reduction rates for each parameter are defined based upon the reduction policy as follows:

Reduction Policy	Range of Values	Default Value
exponential	**[0,1]**	**1**
linear	**[0, ∞)**	**0**

For example, setting the reduction rate of **epsilon** while using a **linear** reduction policy to **5** proceeds as follows:

```
>indifferent-selection --reduction-rate epsilon linear 5
```

A call to the **reduction-rate** switch with a parameter name and reduction policy, but no reduction rate, will return the current reduction rate for the parameter in the reduction policy. For example, after the command above:

```
>indifferent-selection --reduction-rate epsilon linear
5
```

As an example of reduction policies, consider the following sequence of commands. Assume that an agent has been pre-loaded using the **source** command, and the agent has been initiated and is in the proposal phase:

```
>indifferent-selection --epsilon 0.5
>indifferent-selection --reduction-policy epsilon exponential
>indifferent-selection --reduction-rate epsilon exponential 0.9
>step
>indifferent-selection --epsilon
0.45
>step
>indifferent-selection --epsilon
0.405
```

Note that after each decision phase the value of the **epsilon** exploration rate reduces exponentially by a factor of **0.9**.

Many agents will not require automatic reduction of exploration parameters (or may require greater degrees of flexibility/customization). Thus, the **indifferent-selection** command with the **auto-reduce** switch controls this functionality:

```
indifferent-selection [-a|--auto-reduce] <setting>
```

The **setting** parameter can be either **on** or **off**. A call to the **auto-reduce** switch without a **setting** parameter will output the current automatic policy parameter reduction setting. For example:

```
>indifferent-selection --auto-reduce
off
>indifferent-selection --auto-reduce on
>indifferent-selection --auto-reduce
on
```

Note that for performance purposes, the default setting for **auto-reduce** is **off**.

For convenience, calling the **indifferent-selection** command with the **stats** switch will output a complete set of exploration policy information. For example, assuming defaults and Soar-RL is enabled:

```
>indifferent-selection --stats
Exploration Policy: epsilon-greedy
Automatic Policy Parameter Reduction: off

epsilon: 0.1
epsilon Reduction Policy: exponential
epsilon Reduction Rate (exponential/linear): 1/0

temperature: 25
temperature Reduction Policy: exponential
temperature Reduction Rate (exponential/linear): 1/0
```

7.2. Preference Updates

Soar-RL does TD-learning: the estimated Q-value at time **t**, $\mathbf{Q(s_t,a_t)}$, is updated in the direction of a later estimate of this quantity. For the **sum** reward accumulation mode, the Q-value update, where α is the learning rate, is the following equation:

```
Current Estimate += α(Target Estimate – Current Estimate)
```

The update is computed in decision phase **n+1** and then is apportioned to the Soar-RL rules that fired for the operator selected in decision phase **n** in a least mean squares (LMS), gradient-descent fashion.

Updating a Soar-RL rule involves changing the value of its numeric preference. For example:

1. In decision phase **n**, Soar-RL rules **rl-1** and **rl-2** fire for operator **O2**. They have the following numeric preferences:

   ```
   rl-1: (<s> ^operator <o> = 2.3)
   rl-2: (<s> ^operator <o> = -1)
   ```

2. **O2** is selected.
3. In decision phase **n+1**, update **0.2** is computed.
4. **rl-1** and **rl-2** are updated with the following numeric preferences:

   ```
   RL-1: (<s> ^operator <o> = 2.4)
   RL-2: (<s> ^operator <o> = -0.9)
   ```

   Note that the update value is divided amongst and applied equally to all contributing numeric preferences originating from Soar-RL rules.

7.2.1. Target Estimate Calculation

The target estimate is the result of applying discounting to accumulated reward. The discount factor (γ), configured using the **discount-rate** parameter, allows the agent to value immediate rewards over more distant rewards. The value that is chosen for discount is configured by the **learning-policy** parameter.

7.2.2. Learning Calculation

The learning rate (α), configured using the **learning-rate** parameter, dictates the speed by which updates affect agent behavior.

7.2.3. Update Calculation

With respect to intermediate calculations, the final update is calculated as follows:

$$\text{Current Estimate} = Q(s_t, a_t)$$
$$\text{Next Value} = F(\text{learning-policy, t, Q-function})$$
$$\text{Target Estimate} = r_{t+1} + \gamma(\text{Next Value})$$
$$\text{Update} = \alpha(\text{Target Estimate} - \text{Current Estimate})$$

The **update** is then applied as described above.

7.3. Gaps in Rule Coverage

Since TD updates are transmitted backwards through the stored Q-function, it is tempting to think that the function must be well-represented by Soar-RL rules at each decision cycle: if there are no Soar-RL rules to fire for the operator at step N in your task, then the steps prior to N will never receive updates; if there are Soar-RL rules at step N, but they are overly general, then the steps prior to N will receive inaccurate updates.

However, needing a Q-value stored for every decision implies that Soar-RL rules must be provided even for operator selections that are to be decided by symbolic preferences. Maintaining this level of discipline can be difficult for agent programmers, particularly when operators are required that do not represent steps in a task, but perform management of working memory. For example, inserting an operator simply to remove some o-supported structure requires providing Soar-RL rules, sufficiently specific Soar-RL rules to distinguish different Q-values for all the states in which it may be selected.

To address this practical issue, Soar-RL provides preliminary support for automatic propagation of updates over "gaps." A gap is defined as one or more contiguous decision cycles during which no Soar-RL rules fire. By default, Soar-RL will automatically propagate updates over gaps, discounted exponentially by the **discount-rate** parameter with respect to the length of the gap (defined as the number of decision cycles). This behavior can be enabled/disabled by manipulating the **temporal-extension** parameter. If the **temporal-extension** parameter is set to **off**, no updates will propagate across gaps. For tools in identifying gaps, see 10.1. Trace Information.

7.4. Eligibility Traces

By keeping a trace of state-action pairs encountered, the agent can update Q-values for these stored pairs based on a combination of multi-step targets. This can speed learning, particularly when the reward horizon is long. Multi-step updates are averaged together according to the **eligibility-trace-decay-rate** (λ) parameter and discounted according to the **discount-rate** (γ) parameter.
Eligibility traces are implemented by keeping (Soar-RL rule, eligibility trace) pairs for all RL rules with non-negligible traces (as defined by the **eligibility-trace-tolerance** parameter). Memory usage for eligibility traces is minimal when the **eligibility-trace-decay-rate** parameter is set to 0, and grows with increase in the parameter, but is never larger than **O**(# of Soar-RL rules).

The eligibility trace implementation Soar-RL uses depends upon the current value of the **learning-policy** parameter:

Learning Policy	Eligibility Trace Implementation
sarsa	Sarsa(λ)
q-learning	*Watkin's* Q(λ)

The Sarsa(λ) algorithm is described below as implemented in Soar-RL. Assume that **e** represents the list of (Soar-RL rule, eligibility trace) pairs for all non-negligible traces (initially empty), **o** is an operator associated with a Soar-RL rule, **n** represents the number of Soar-RL rules that contributed to the selection of **o**, and **Q** represents the numeric preference values associated with Soar-RL rules (accessed by operator id).

```
Initialize reward r = 0
Repeat (for each Soar cycle):

        Select operator o

        Repeat (for each o in e):
                if ( e[o] < [eligibility trace tolerance ] )
                        remove e[o]

        increment = 1/n

        if ( e[o] )
                e[o]+=increment
        else
                e[o]=increment

        Repeat (for each o in e):
                Q[o]+=(learning rate)•(update)•e[o]
                e[o]*=(discount rate)•(eligibility trace decay rate)

        Apply operator o
        Observe reward r
```

The *Watkins* Q(λ) algorithm is equivalent to Sarsa(λ) in terms of calculation. However, if the selected operator, **o**, was not the proposed operator with greatest combined numeric preference values (i.e. the "greedy" choice), all eligibility traces are re-initialized to zero.

In the calculations above, an **eligibility-trace-decay-rate** parameter set to **0** will result in eligibility trace values always being removed, and thus Q-values never decaying.

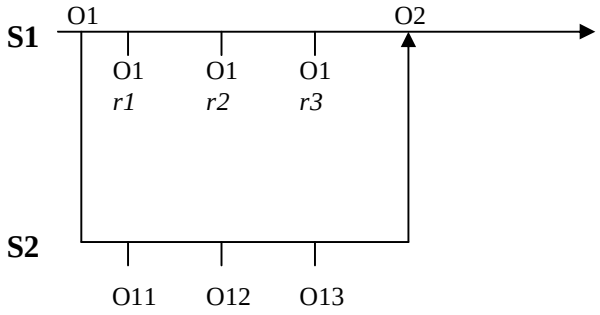
Consequently, Soar-RL interprets such a setting (the default) as disabling eligibility traces.

7.5. Hierarchical Learning

Hierarchical reinforcement learning (HRL) is reinforcement learning done over a hierarchically decomposed task structure. Learning can be applied both to improving the implementation of a subtask and to the selection among subtasks. Compared to flat RL, HRL can demonstrate faster learning on a single task and can learn policies that are easier to transfer to related tasks.

7.5.1. Operator No-Change Impasses

Hierarchical reinforcement learning in Soar-RL is built on Soar's operator no-change impasse, which has traditionally been used for subgoaling and hierarchical task decomposition. Consider the following operator trace where operator **O1** is selected and impassed at state **S1**, with operators **O11**, **O12**, and **O13** being selected and applied at substate **S2**:



Soar-RL will treat the operator selection at **S1** and **S2** as two separate RL problems (**S1** and **S2** have independent **reward-link** structures):

S1: Rewards at **S1** while **O1** is impassed are attributed to **O1**. By default, these rewards and the next-state prediction are discounted by the number of decision cycles that **O1** has been impassed. So if rewards **r1**, **r2**, and **r3** are the rewards received at **S1** while **O1** is impassed, the target estimate for **Q(S1, O1)** is

$$\mathbf{r1} + \gamma(\mathbf{r2}) + \gamma^2(\mathbf{r3}) + \gamma^3[\ \mathbf{Q(S1, O2)}\ ]$$

This model maintains the definition of the Q-function as representing the expected discounted sum of future reward received after selecting an operator.

Setting the **hrl-discount** parameter to **off** will change this behavior, such that the number of cycles **O1** has been impassed will be ignored. Thus the target estimate for **Q(S1, O1)** would be

$$\mathbf{r1} + \mathbf{r2} + \mathbf{r3} + \gamma[\ \mathbf{Q(S1, O2)}\ ]$$

S2: After applying **O13**, immediately before the state **S2** is removed, the architecture checks for reward **r** at **S2**. The target estimate for **Q(S2, O13)** is just **r**.

7.5.2. Other Soar Impasses

For impasses other than operator no-change, the behavior of Soar-RL at top-state **S1** and substate **S2** is as follows:

S1: During these impasses, there is no operator installed at **S1**. If **O1** is the last operator selected before the impasse, **r** the reward received in the decision cycle immediately following **O1**, and **O2** the first operator selected after the impasse, then **O1** is updated with the target **r** + γ[**Q(S1, O2)**]. In other words, Soar-RL acts as if the impasse hadn't occurred.

S2: Soar-RL acts exactly as it does for an operator no-change: the substate is treated as an episodic task.

Note: in the final version of Soar-RL, all Soar impasses will be treated identically. In the case of an impasse other than operator no-change, the time period at the superstate during which no operator is selected will be treated as a "gap" in rule coverage, and perceived reward will be discounted with respect to decision cycles passed.

8. Soar-RL Parameters

Soar-RL is configured using the **rl** command.

8.1. Parameter Configuration

Individual configuration parameters are retrieved and manipulated using the **get** and **set** switches of the **rl** command:

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

Agents can retrieve and change parameters in the actions of rules using the **cmd** function.

8.2. Parameter Descriptions

8.2.1. General

Soar-RL

Purpose	Enable or disable Soar-RL	
Parameter	**learning**	
Values	**on**	Enable Soar-RL
	off	Disable Soar-RL
Default	**off**	

Temporal Extension

Purpose	Direct how Soar-RL should behave during gaps in Soar-RL rule coverage	
Parameter	**temporal-extension**	
Values	**on**	Automatically propagate discounted updates
	off	Do not propagate updates across gaps
Default	**on**	

8.2.2. Reward Discount

Discount Rate

Purpose	Set the discount rate (γ)
Parameter	**discount-rate**
Values	Numeric, **[0,1]**
Default	**0.9**

8.2.3. Learning

Learning Rate

Purpose	Set the learning rate (α)
Parameter	**learning-rate**
Values	Numeric, **[0,1]**
Default	**0.3**

Learning Policy

Purpose	Modify the type of TD-learning used by Soar-RL	
Parameter	**learning-policy**	
Values	**sarsa**	On-policy, *SARSA(λ)* – update from the next action selected, target $r_{t+1} + \gamma \cdot Q(s_{t+1}, a_{t+1})$
	q-learning	Off-policy, *Q-learning* – update from the next candidate action with best estimate, $r_{t+1} + \gamma \cdot \max_a [Q(s_{t+1}, a_{t+1})]$
Default	**sarsa**	

HRL Discount

Purpose	Direct how Soar-RL should discount reward at impassed states	
Parameter	**hrl-discount**	
Values	**on**	Reward received at an impassed state is discounted by the number of cycles of impasse
	off	Reward is not discounted by the number of cycles of impasse
Default	**on**	

8.2.4. Eligibility Traces

Eligibility Trace Decay Rate

Purpose	Set the eligibility trace decay rate (λ)
Parameter	**eligibility-trace-decay-rate**
Values	Numeric, **[0,1]**
Default	**0**

Eligibility Trace Tolerance

Purpose	Sets the level at which eligibility traces are considered negligible
Parameter	**eligibility-trace-tolerance**
Values	Numeric, **(0, ∞)**
Default	**0.001**

8.3. Full Parameter Configuration

Entering simply the **rl** command (with no switches) will return full parameter configuration information. For example, assuming default configuration, the result of executing **rl** is as follows:

```
rl
Soar-RL learning: off
temporal-extension: on

Discount
--------
discount-rate: 0.9

Learning
--------
learning-policy: sarsa
learning-rate: 0.3

Eligibility Traces
------------------
eligibility-trace-decay-rate: 0
eligibility-trace-tolerance: 0.001
```

8.4. Parameter Behavior

This section details two fundamental characteristics of parameter behavior: parameter configuration timing and handling of invalid parameter values. Additionally, there are some special cases to discuss.

8.4.1. Parameter Configuration Timing

In order for parameter configuration to affect the behavior of Soar-RL during cycle **n**, parameter configuration must be completed before the start of the *decision* phase of cycle **n**.

8.4.2. Invalid Parameter Values

Upon attempting to **set** a Soar-RL parameter, the new value is validated. If the value is found to be invalid, the system will use the previous value.

8.4.3. Special Cases

At the beginning of the *decision* phase of each cycle, the value for the **learning-policy** parameter is checked. If this value has changed since the last cycle, all eligibility traces on the current state are re-initialized.

9. Soar-RL Statistics

Feedback from the Soar-RL system is retrieved using the **stats** switch of the **rl** command:

```
rl [-S|--stats] <statistic>
```

If a **statistic** argument is provided, the command returns the value of a specific statistic. The valid **statistic** arguments are listed below.

Statistic	**update-error**
Description	The difference between the **target estimate** and the **current estimate** in the most recent update (see *7.2.3. Update* Calculation).

Statistic	**total-reward**
Description	The total accumulated reward (with respect to the **accumulation-mode**) in the most recent update (see *7.2.1. Target* Estimate Calculation).

Statistic	**global-reward**
Description	The total accumulated reward (with respect to the **accumulation-mode**) since agent initialization (see *7.2.1. Target* Estimate Calculation).

Agents can retrieve specific statistics in rule actions using the **cmd** function.

Entering the **rl --stats** command with no **statistic**, or an invalid **statistic**, will return all statistics. A sample execution may look as follows:

```
>rl --stats
Error from last update: 0.8
Total reward in last cycle: 7
Global reward since init: 51
```

10. Trace and Command Information

This section details debugging tools, including trace information, print switches, excise switches, and decision cycle commands.

10.1. Trace Information

Viewing of numeric preferences for each operator can be accomplished using the following watch switch:

```
watch [-i|--indifferent-selection]
```

This watch function is not enabled by default or through any watch level. Use of the **preferences** command provides further information about operators referenced with this trace.

To view firing of templates, use the following watch switch:

```
watch [-T|--template]
```

This watch function is not enabled by default, but is enabled at watch level three (3).

To view Soar-RL debugging information, use the following watch switch:

```
watch [-R|--rl]
```

This function is not enabled by default or through any watch level. At present, this watch level provides trace information about starting and ending of gaps in Soar-RL rule coverage.

10.2. Print Switch

Soar-RL introduces two new switches to the **print** command:

```
print [-r|--rl]
print [-T|--template]
```

The **rl** switch to the **print** command provides detailed information about all Soar-RL rules. The output displays the rule name, the number of updates (potentially fractional, if updated by an eligibility trace), and the current value. For example, if **my*reinforcement*learning*rule** were the only Soar-RL rule in an agent and had been updated 4 times with a current value of 3.7, the result would be as follows:

```
>print --rl

my*reinforcement*learning*rule      4.    3.7
```

The **template** switch to the **print** command provides the names of all template rules. For example, if **sample*template*rule** were the only Soar-RL template rule in an agent, the result would be as follows:

```
>print --template

sample*template*rule
```

Both new switches adhere to existing **print** command switches. For instance, to view information about a specific rule or template, simply indicate its name as a parameter. Also, to view the complete rule, use the **full** switch. The following examples illustrate some common usages:

```
>print --rl --full

sp {my*reinforcement*learning*rule
    (state <s> ^operator <o> +)
-->
    (<s> ^operator <o> = 2.3)
}


>print --template --full

sp {sample*template*rule
    :template
    (state <s> ^operator <o> +
               ^value <v>)
-->
    (<s> ^operator <o> = 2.3)
}


>print --name my*reinforcement*learning*rule

my*reinforcement*learning*rule      4.    3.7


>print --full sample*template*rule

sp {sample*template*rule
    :template
    (state <s> ^operator <o> +
               ^value <v>)
-->
    (<s> ^operator <o> = 2.3)
}


>print --all

my*reinforcement*learning*rule      4.    3.7
sample*template*rule
```

10.3. Excise Switch

Soar-RL introduces two new switches to the **excise** command:

```
excise [-r|--rl]
excise [-T|--template]
```

The **rl** switch to the **excise** command removes all Soar-RL rules (including those created from templates) from production memory. For example, if **my*rl*rule** were the only Soar-RL rule in an agent, the result would be as follows:

```
>excise --rl

1 production excised.

>print --rl
```

The **template** switch to the **excise** command removes all Soar-RL templates, preventing further creation of new Soar-RL rule instantiations. For example, if **my*rl*template** were the only template rule in an agent, the result would be as follows:

```
>excise --template

1 production excised.

>print --template
```

10.4. Decision Cycle Commands

Soar-RL introduces two new commands to debug the decision cycle of an agent:

```
predict
select <id>
```

The **predict** command determines, based upon current operator proposals, which operator will be chosen during the next decision phase. If **predict** determines an operator tie will be encountered, **tie** is returned. If **predict** determines no operator will be selected (state no-change), **none** is returned. If **predict** determines a conflict will arise during the decision phase, **conflict** is returned. If **predict** determines a constraint failure will occur, **constraint** is returned. Otherwise, **predict** will return the id of the operator to be chosen. If operator selection will require probabilistic selection, and no alterations to the probabilities are made between the call to **predict** and decision phase, **predict** will manipulate the random number generator to enforce its prediction.

The **select** command will force the selection of an operator, whose id is supplied as an argument (case-insensitive), during the next decision phase. If the argument is not a proposed operator in the next decision phase, an error is raised and operator selection proceeds as if the **select** command had not been called. Otherwise, the supplied operator will be selected as the next operator, regardless of preferences. If **select** is called with no **id** argument, the command returns the operator id currently forced for selection (by a previous call to **select**), if one exists.

11. Soar-RL Programmer Reference

The following tables list basic information about each of the Soar-RL related commands. It is not intended to substitute for this document, but a quick reference for commonly used commands and options.

11.1. Soar-RL

Useful Commands

Command	Description
`rl`	Summary table of parameter settings
`rl [-g|--get] <parameter>`	Retrieve a Soar-RL parameter value
`rl [-s|--set] <parameter> <value>`	Set a Soar-RL parameter value
`rl [-S|--stats] <statistic>`	Access Soar-RL statistics
`print [-r|--rl]`	Print Soar-RL rules
`print [-T|--template]`	Print Soar-RL templates
`watch [-R|--rl]`	Soar-RL debugging trace
`watch [-T|--template]`	Soar-RL template firing trace
`excise [-r|--rl]`	Excise Soar-RL rules
`excise [-T|--template]`	Excise Soar-RL templates

Soar-RL Parameters

General

Parameter Name	Acceptable Values	Default
`learning`	`on` `off`	`off`
`temporal-extension`	`on` `off`	`on`

Reward Discount

Parameter Name	Acceptable Values	Default
`discount-rate`	`[0,1]`	`0.9`

Learning

Parameter Name	Acceptable Values	Default
`learning-rate`	`[0,1]`	`0.3`
`learning-policy`	`sarsa` `q-learning`	`sarsa`
`hrl-discount`	`on` `off`	`on`

Eligibility Traces

Parameter Name	Acceptable Values	Default
`eligibility-trace-decay-rate`	`[0,1]`	`0`
`eligibility-trace-tolerance`	`(0,∞)`	`0.001`

11.2. Operator Selection

Useful Commands

Command	Description
`indifferent-selection [-s|--stats]`	Summary of settings
`indifferent-selection`	Current exploration policy
`indifferent-selection <policy>`	Set exploration policy
`indifferent-selection <parameter> <value>`	Get/Set exploration policy parameters
`predict`	Predict the next selected operator
`select <id>`	Force the next selected operator

Exploration Policies

Policy Name	Description
`[-b|--boltzmann]`	Tempered softmax (uses temperature)
`[-g|--epsilon-greedy]`	Tempered greedy (uses epsilon)
`[-x|--softmax]`	Random, biased by numeric indifferent values
`[-f|--first]`	Deterministic, first indifferent preference is selected
`[-l|--last]`	Deterministic, last indifferent preference is selected

Exploration Parameters

Parameter Name	Acceptable Values	Default
`[-e|--epsilon]`	`[0,1]`	`0.1`
`[-t|--temperature]`	`(0,∞)`	`25`