New Goal System Quick Start Guide

Last updated 26 April 2012

NGS is a lightweight "declarative goals" implementation of a Forest of Goals approach to Soar programming. NGS provides ways to manage goals and their relationships on the top state, versus through Soar's state stack. NGS is implemented as a set of Tcl macros and associated Soar code for goal management. LHS macros create bindings that can be used in the RHS of a production, and RHS macros create or manipulate objects. Like other APIs, it is encouraged that developers using NGS use the LHS and RHS macros provided for creation and access to maintain this encapsulation.

This document gives a very brief introduction to the most commonly used features of NGS, as well as an very brief example at the end. NGS also has a host of convenience macros for goals, operators and the like, but those are not described here. See "NGS User Guide" for more complete information about the New Goal System.

A note about notation.

Most macros have a set of required and optional parameters. We use the Tcl notation for macros here, so an optional argument to a macro will appear in embedded two-tuple brackets, with the argument identifier given first and default value appearing second, as in:

```
     NGS_match-active-goal { goal_type {goal_bind ""} {state_bind <s>} }
```

Here {state_bind <s>} shows an optional argument called state_bind to the NGS_goal-creation macro. When the argument is not filled in the procedure call, <s> is used as the value by default. In some cases, the empty string ("") may be given as the default value, which results in a binding variable to be auto-generated by the system. This will mean it is only used internally to the macro and you cannot use it. We may give a simplified form of a macro here to hide some of the potentially more complex uses of the macro. Where this is done, a marker (†) is given to show that there is a more complex form available.

Objects, Tags, and Types

Objects and Tags

NGS has an object system that allows for the definition of objects. The only real differences between a regular Soar structure and an NGS object is the insertion of a "tags" structure inside that object, and two such tags have to do with the type of that object, as defined by the developer.

Tags are a collection of state or routine progress markers on the object, such as for its type, its state, etc. There are also various access routines for setting and getting things within the tags structure. (Note that you need to bind to the tags substructure on a goal as part of the LHS of a production using an appropriate macro. Typically the "*tags"* binding is an optional argument.)

To create an NGS object, use this macro:

```
      NGS_create-object { parent attribute
                          object_type_list object_bind {tags_bind ""} }

      Example: NGS_create-object <parent> <child> childobject <obj> <tags>
```

After an object is created, you can generically set the tag on an object using the *tags_bind* that was added during object creation:

```
NGS_tag {tags_bind tag_name tag_value}

Example: NGS_tag <tags> completed *yes*
```

To test if an object has a tag or does not have a tag with a given name (and optional value):

```
NGS_is-tagged { object_id tag_name {tag_val "" } }
NGS_is-not-tagged { object_id tag_name {tag_val "" } }

Example: NGS_is-tagged <obj> completed *yes*
```

Since a common attribute on an object is its name (that is, many objects are created with a "name" tag), there is a special macro just for testing the name:

```
NGS_is-named { object_id name }
```

Types

Since there is not an actual typing system available in Soar, types are simply string values associated with an object that might allow for specific behavior specialization by this tag. Each created NGS object will also have a "most-derived-type" tag that indicates the most **specific** type associated with the object. By testing for the object type (rather than the most-derived-type), you can effectively use an object as an instance of a superclass object. An object's type is set through the object creation procedure:

```
NGS_create-object { parent attribute
                    object_type_list object_bind {tags_bind ""} }
```

Here an important attribute is "object type list" which is a semi-ordered list of types which are characteristic of this object, with the very last type in the list being implicitly defined as the objects most-derived-type.

```
Example: NGS_create_object <tstate> obstacle {obstacle barrel}
                           <obstacle>
```

Note that in the simplest case, the object_type_list parameter can simply be a string defining the object's type.

Simplified form:

```
† NGS_create-object { parent attribute object_type
                      object_bind }

Example: NGS_create_object <tstate> obstacle barrel <obstacle>
```

The type can be referenced by NGS macros for testing the type of the object:

```
NGS_is-type { object_id type_name }
NGS_is-not-type { object_id type_name }
NGS_is-most-derived-type { object_id type_name }
NGS_is-not-most-derived-type { object_id type_name }

Example: NGS_is-most-derived-type <obstacle> barrel
```

The matches of *NGS_is-most-derived-type* will always be a subset of the matches from *NGS_is-type*, for the same type parameter. Note that if you're not using complex types, then *NGS_is-type* and *NGS_is-most-derived-type* are equivalent.

Implemented Objects: Goals
Goals encapsulate the state associated with some behavior, and can be implemented across a number of operators. Goals have a type (typically the name of the operator), stored in the ^most-derived-type attribute. Goals also have a "status" stored in the tags (active, suspended, terminated) that says where they are in their execution cycle. Goals can be i-supported, which makes clean up free and easy. Terminated o-supported goals are removed after a fixed time. (This is set in the parameter `NGS_UNCONSIDERED_ITEM_REMOVAL_TIMEOUT`. If you wish to prevent this timeout, excise the production "`ngs*terminated-considerable*set-timeout`" after loading NGS.)

Since NGS is primarily about goal management, goals are treated in more detail through the rest of this document.

Goals

You can create goals as i-supported or o-supported, depending on how you create them (just as with any normal Soar object): either in an elaboration, or in an operator application.

Before creating a goal, you need to get a binding to the desired goal pool, using this LHS macro:

```
NGS_match-goalpool { goal_pool_bind {state_bind <s>} }
```

Simplified form:

```
† NGS_match-goalpool { goal_pool_bind }

Example:    NGS_match-goalpool <pool>
```

From this, you can then create a goal, using this RHS macro:

```
NGS_create-goal {goal_pool goal_type_list {goal_bind ""} {tags_bind ""} }
```

Simplified form:

```
† NGS_create-goal {goal_pool goal_name {goal_bind ""}
```

Note also that this creates the goal in the *desired* pool, and takes a decision cycle to transform this to an *active* goal (via NGS productions).

Binding to a Goal
To do things with a created goal, you need to get a match on a goal before using it. Here, test for an existing (active) goal:

```
NGS_match-active-goal { goal_type {goal_bind ""} {state_bind <s>} }
```

Simplified form:

```
† NGS_match-active-goal { goal_type {goal_bind "" }}
```

```
Example: NGS_match-active-goal execute-mission <g>
```

Creating subgoals

If you have a goal, you might want to create a subgoal for it. Similarly, this is an RHS macro.

```
NGS_create-subgoal {goal_pool goal_type_list
                    super_bind {goal_bind ""} {tags_bind ""} }

Example: NGS_create-subgoal <pool> execute-mission <parent> <newgoal>
```

Note also that this creates in the *desired* goal pool, and takes a decision phase to move this to the *active* goal set.

Testing Goal Relationships

Once we have some goals, we can do some tests on their relationships. These both return true or false, and can be used on the LHS of productions.

```
NGS_is-parent-goal { goal supergoal {supergoal_type ""} }
NGS_is-subgoal { goal subgoal {subgoal_type ""} }
```

Simplified forms:

- † *NGS_is-parent-goal { goal supergoal }*
- † *NGS_is-subgoal { goal subgoal }*

Setting and Testing Status

Goal state can be set with the following RHS macros that take the goal's tags structure as an argument (so you must grab the *tags_bind* on the LHS before calling these macros).

```
NGS_mark-active { tags_bind }
NGS_mark-suspended { tags_bind }
NGS_mark-achieved { tags_bind }
NGS_mark-unachievable { tags_bind }
```

Testing for the status of a goal on LHS can be done on a bound goal:

```
NGS_is-active { goal_bind }
NGS_is-suspended { goal_bind }
NGS_is-achieved { goal_bind }
NGS_is-unachievable { goal_bind }

Example: NGS_is-active <mygoal>
```

Using Operators

Operators are how to do something to accomplish goals. NGS encourages a declarative associate between operators and goal. With this RHS macros, we can propose an operator in reference to a goal:

```
NGS_create-operator { operator_type_list
                      goal_id
                      {operator_bind "<o>"}
                      {operator_pref_list "+ ="}
                      {state "<s>"} }
```

Note this takes the last element of the *operator_type_list* as the name of the operator, and allows the developer to set preferences on the operator at creation, with the default being "+ =".

Simplified form:

```
    † NGS_create-operator-from-goal { operator_name goal_id <o>}
```

```
    Example: NGS_create-operator fly-orbit <g> <o>
```

Once we have the operator, we want to apply it. With NGS, this means getting operator bindings via an LHS macro (with optional bindings to its related goal):

```
    NGS_match-operator {operator_name {operator_bind ""}
                       {goal_bind ""} {goal_tags_bind ""} {state_bind "<s>"} }
```

Simplified form:

```
    † NGS_match-operator { operator_name {operator_bind ""} }

    Example: NGS_match-operator create-mission-subgoal <o>
```

Note that if you want to do something with the operator (such as change its attributes), you need to create an explicit binding, hence the <o> in the example calls. If you want to do something with its related goal (such as marking it achieved) then you need to pass a binding for the goal and tags.

Example

Here is an example set of productions that exercise several of the macros described above. Note that since these use Tcl macros, all productions use quotes ("…") instead of curly braces.

```
# create the goal
sp "hello-world*create-goal
   -(<s> ^hello-world-done *true*)
    [NGS_match-goalpool <goals>]
-->
    [NGS_create-goal <goals> say-hello-world]"

#propose the operator
sp "hello-world*propose-operator
    [NGS_match-active-goal say-hello-world <g>]
    [NGS_is-not-tagged <g> complete *yes*]
    (<g> ^tags <tags>)
    (<tags> -^complete <any-val>)
-->
    [NGS_create-operator say-hello-world <g> <o>]"

# apply the operator
sp "hello-world*apply
     [NGS_match-operator say-hello-world <o> <goal> <gtags> <s>]
-->
     (write (crlf) |HELLO!|)
     (<s> ^hello-world-done *true*)"
```