New Goal System User's Guide

Last modified: 12 June 2012

Table of Contents

1 Introduction to the New Goal System

1.1 What is NGS?

The New Goal System (NGS version 3) is a Soar library to support faster development of Soar agents by allowing programming at a higher level of abstraction than that allowed by using Soar alone. NGS uses Tcl macros and added Soar productions to help implement these concepts within a Soar agent. Through these macros, NGS incorporates elements of encapsulation to hide many of the low-level Soar details of object creation, and provides interfaces for matching to standard object structures such as goals and operators.

Technically, NGS has three interacting components:

1) a set of knowledge structures and procedures implementing the Forest of Goals (FoG) design pattern in Soar code
2) a set of Tcl macros constituting an API that support common FoG usage patterns (e.g., object creation, goal manipulation, Left-Hand-Side match templates, etc.)
3) A set of optional modules that support various extensions to FoG, such as other design patterns that use FoG as a baseline. (There are no released modules for the base release 3.0)

NGS requires Tcl as a preprocessor for Soar productions. Tcl macros are expanded into Soar code at load time. The macros include templates for portions of LHSs and RHSs of productions, called *fragments* or *manipulators*, as well as macros for generating entire productions or sets of productions. The current version of Soar from the University of Michigan ("c-Soar," v9.3 as of this writing) does not support Tcl, but this can be worked around[1]. The Java implementation of Soar, jSoar, supports Tcl directly.

This document describes the basic concepts in NGS, details about how it is implemented and how it is intented to be used, and the history of its development. Other reference documentation exists elsewhere, including the "NGS Quick Guide" as a way to get rolling with NGS quickly.

1.2 The history and philosophy of NGS

NGS came about to explore an alternative approach to Soar development from the standard "Michigan Approach." In particular, NGS implements a Forest of Goals approach that allows for multiple goal trees (a "forest") represented using declarative goal structures. These goals can affect the operator stack, but they are represented externally to that stack. This is in contrast to the "Michigan Approach" which represents goals implicitly in the preconditions of operator proposals, and maintains only a single stack of goals represented in the state/operator stack supported directly by Soar. In many Soar-based systems, we found that we were building systems that effectively had to

[1] See *tcltosoar.tcl* converter (part of this package) to pre-process Tcl macros in C-soar Soar 9.3.

manage multiple simultaneously goals, but there was no architectural support for this. NGS was created to help make this process easier.

NGS implements a *forest of goals* using dedicated data structures to support those goals, their relationships, and their management. It does not replace Soar programming, but rather augments it. It accomplishes this through a library of Soar rules and Tcl macros that try to maintain a very small footprint. However, its has some potential drawbacks to be aware of.

First, as with any Soar program, NGS is not fully encapsulated; it depends on naming conventions for productions and memory structures for encapsulation. This type of encapsulation is a double-edged sword: because NGS is implemented in Soar and uses Soar's working memory, Soar code written by a programmer can see and affect structures that are internal to NGS' workings. Proper use of NGS requires that a developer not use reserved NGS structures inappropriately, and not use the same naming scheme for Soar productions as NGS does (see details below). Otherwise, a programmer could inadvertently clobber NGS and make it behave unexpectedly. On the other hand, this loose encapsulation eases domain-specific expansion or plug-in development, and also allows meta-cognition on the part of the agent, where a Soar agent can consider and adjust its own NGS-related knowledge structures (goals, for instance). In rare cases, it may even be desirable to overwrite an NGS rule with one customized for your application.

Second, because NGS is itself written using rules, NGS processing happens in the match and fire cycles that your own system's rules operate in. The underlying NGS implementation does not use operators, so it will not interfere with your own operator selection or preferences. But since NGS is implemented as rules, NGS could potentially affect timing in highly tuned rule systems. However, typical use of NGS is practically invisible to a Soar agent's operation.

Third, NGS can make debuging a little less straightforward. There are NGS rules that will appear in any list of rules, which may be unfamiliar to someone writing their own rules (e.g., in a "print" or "matches" command). By convention, these are prefixed by "ngs" so they could be filtered out. Also, due to Tcl macros expansion, the rules that are generated do not exactly match the rules that you would see in the original source code. Printing a production from a Soar agent window shows fully expanded productions rather than the source productions containing Tcl macros[2].

Despite these downsides, using NGS and Tcl macros can help make your agent development faster and more straightforward.

[2] One trick which can show you the expanded version of a production is to replace the "sp" in front of the production with an "echo" command. When you do this in the NGS namespace, the production's expanded form will show in the agent window. Example: namespace eval NGS {echo "my-production*foo (<s> ^superstate nil)"}

2 NGS Concepts – Macros, Objects and Type Information

2.1 Macros

Much of NGS is implemented as TCL macros which are invoked by the developer as part of writing Soar rules. These macros are interpreted when the productions are loaded into a Soar agent (or in a pre-processing step using the *tcltosoar.tcl* converter, as mentioned earlier). Since the TCL interpreter checks all loaded Soar code, the end-user can also define their own macros.

When the TCL code is interpreted at production-load time, each macro is called and returns some text string (usually an expanded version of its inputs). That text string is then interpreted as part of the Soar production.

Any errors in the use of the TCL macros (or in the macros themselves) will become apparent at this point – an error message will be printed and the agent will *stop loading Soar rules*. NGS stores information both in Soar productions and in the TCL interpreter. This means that if, during development, you correct an error in a macro or its use, it is best to completely restart the agent instead of attempting to reload productions.

2.2 Objects and Typing

In NGS, there are a few important categories of data structures:

- Object – any structure for storing knowledge, with a declared type (possibly with a type hierarchy) and specially named substructure (Tags, below). This is the most basic category. Nothing in NGS actually has the type "Object", but some of the other types are informally considered to be "Objects".
- Goal – a representation of a problem solving context. All concrete goal instances inherit from this class. Goals are a kind of "considerable" object internally to NGS; considerables are objects that the system can reason about before doing anything substantive with them.
- Tags – a reserved sub-structure on all objects that is used to track the object's state or other process markers. Tags are *not* objects, though they are attached to objects.
- Objects – these are unchanged in NGS from typical Soar use, but are noted here to describe their relationship to goals

Figure 1 below illustrates the relationship between these NGS object types.

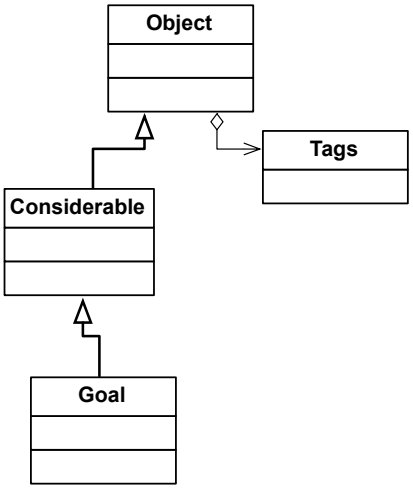

Figure 1: Basic NGS Object Hierarchy

All NGS objects have a type, and may have a name. The type info defines how they function (and is similar to an object's *class* in object-oriented systems), while their name is a human-readable label that can help programmers and users of NGS-based systems to understand what's occurring. This is in contrast to typical Soar knowledge structures, whose attribute name, or occasionally their ^name tag, is their defining characteristic.

NGS implements a rudimentary typing system where a developer can assign multiple *types* to an object. One of those assigned types is the object's *most-derived-type* – in object-oriented terms, the most concrete class of which this object is an instance. The types assigned to the object which are not most-derived-type refer to classes further up the object hierarchy. For example, suppose "G1" is a object, and suppose its most derived type is "user-reply-message", it's superclass is UserMessage, whose superclass is Message. The types of the object would be:

- user-reply-message (most-derived-type)
- UserMessage
- Message

This allows us to "inherit" aspects of classes (such as behaviors) up the hierarchy, but yet lets us refer to the instance by its most specific type. For example, any object that is derived from Message can be affected by behaviors that trigger off the Message type.

Each of the main NGS data structures is described below.

2.3 Goals

Goals represent some desired state of the world or of the agent's knowledge. A goal is a derivation of the base *Object* class, representing knowledge or activity that an agent wishes to explicitly reason over before resources are committed to that knowledge or activity. (Internally to NGS, Goals are a type of "considerable" – where the system "considers" what do to before doing it. In this manual, we refer only to Goals for

simplicity.) Like the Soar architecture's *operator* object, a Goal may be created and deliberated upon before it begins to affect the rest of the system's behavior.

A good rule of thumb for when to create a goal is: whenever you might want to achieve some state (such as knowing some information or changing something in the world), create a goal to manage each of the different ways that you could go about accomplishing that goal. A goal defines context in which to do problem solving, and operators do the work within that context.

2.3.1 Goal States of Existence

Inherent to the notion of goals is that of its *state* – each Goal is in at least one state of existence:

- Desired – the agent has knowledge of the Goal, but has not necessarily committed resources to work on it. A Goal is Desired until it is Terminated. Desired has two substates, Active and Suspended.
    - Active – the agent has committed resources to work on the Desired Goal
    - Suspended – the agent has (temporarily) suspended work on the Desired Goal
- Terminated – the agent has completed (or otherwise stopped) work on the Goal, and it is no longer Desired

Typically, when a Goal is created, it is by default created in the *desired* state. Moving a Goal to any state (*active, suspended* or *terminated)* is a deliberate activity on the part of the agent. In this system, a Goal that is *active* or *suspended* is still considered as *desired*. By default, Goals in the *terminated* state will be removed after some user-configurable time period. Figure 2 below illustrates the states a Goal can take on, and the transitions it can make.

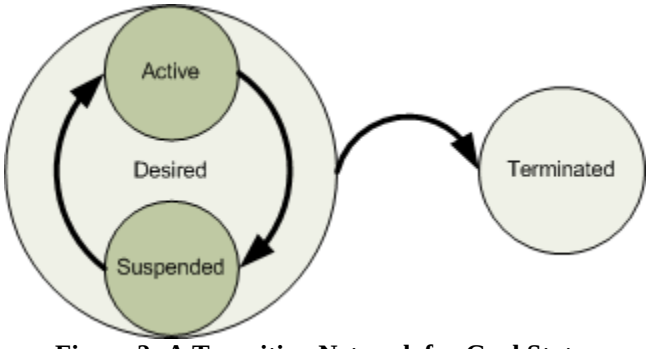

Figure 2: A Transition Network for Goal States

Goals can be created in any of these states; for example, a Goal can be created in a suspended mode, where the Goal can be reasoned about without actually acting on the Goal until later deliberation chooses to do so (by changing the state from suspended to active).

Goals can also be tagged with extra information about why they are in a particular state – for example, Terminated Goals can be marked as *achieved*, *unachievable*, or *aborted.* Once marked with these tags, the goals are automatically moved to the Terminated Pool.

2.3.2 Subgoals

To implement the notion of a Forest of Goals, NGS supports sub-goal and parent-goal relationships among goals. The diagram below illustrates the relationship between multiple goals. A goal can have multiple subgoals, and a subgoal can have multiple parents.

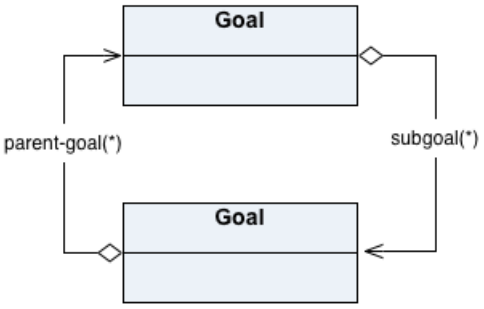

Figure 3: Goals and Subgoals in NGS

Goal state is preserved across subgoal relationships. In particular, the activation of a subgoal with a parent-goal is dependent upon the activation of its parent as well as other factors. For example, if a parent is suspended, any of its active subgoals immediately shift to a suspended state. When the parent goal is unsuspended, its subgoals suspended in this fashion will resume normally. If a parent is achieved, then its subgoals are achieved, etc.

2.4 Tags

Tags are used an organizing structure on an object to store process-related information that does not contribute to the basic definition of the object. For instance, when an object expires, or the time at which some event should be triggered, are treated as tags. This aims to keep the object definition from becoming too cluttered.

2.5 Operators

In the Forest of Goals approaches, the system does not depend heavily upon Soar's native "goal"/operator stack. NGS handles the state information and management of goals in a single location in working memory. Operators, then, become simply a means of choosing among alternatives, setting persistent memory, or placing output commands to the output link.

NGS encourages the use of Goals for contexts in which operators work. This allows better tracing and encapsulation of processes by associating operators with particular

goals. This uses the typical Soar process of requiring that an operator proposal test a state – but in NGS, the goal encapsulates state information explicitly, so an operator tests its parent goal in its proposal. This is illustrated in Figure 4.

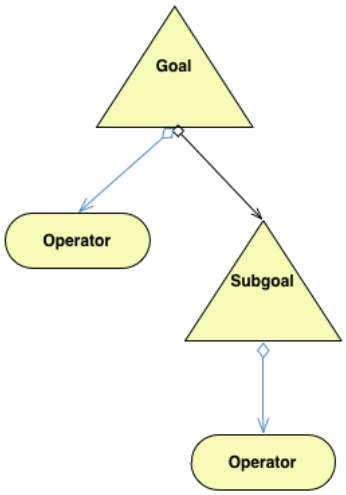

Figure 4: Goal-Operator relationships in NGS

In a pure Forest of Goals implementation, operators do not stay on the stack for more than a single decision cycle: at the end of an operator application cycle, the operator is retracted. To facilitate an approach of not keeping operators on the stack for any amount of time, its is generally the case that an operator should not be proposed unless there is an operator application that can apply – the proposal tests as much as is necessary on its LHS for the applications to be able to fire.

However, you can consider hybrid approaches. There may be conditions under which you may desire an operator to exist on the stack for multiple decision cycles. You might consider an enduring operator to be an indicator of an error state of some kind – e.g., an impasse that indicates lack of knowledge. An operator with impasse-driven subgoaling might (conceptually) also be used if a programmer wished to take advantage of Soar's learning capabilities. We have not yet explored these kind of hybrid implementations in any detail, but theoretically they should be able to be combined. The only thing to note is that NGS currently assumes all NGS goals are stored on the top-state, and this top-state may need to be accessible to other states in a mixed approach.

3 Implementation Details

3.1 Goal States

Goal states are stored in *pools* (sets) of Goals. Desired, Active, and Terminated are set structures in memory, whereby a reference to the goal is placed as a member of one of these sets when that goal is marked with the appropriate state. Other states, namely *Suspended, Achieved, Unachieved, Aborted*, are not represented as distinct memory

pools, but simply as tags. The latter states can be thought of as "substates" or a reason for why the goal is in the particular super-state – e.g., a *Terminated* goal can be marked as *achieved* to indicate that it is terminated because it has been accomplished. In any case, the particular implementation details should not be of concern to a user of NGS, so long as the correct macros are used for testing states and creating objects. These states can be explicitly tested for using LHS fragments. Important things to know about the goal states:

- Standard manipulators (RHS macros) are available to move goals into these pools manually.
- Goals should never be marked both active and suspended, or active and terminated, or terminated and suspended simultaneously. A programmer manipulating Goals via manual tagging should never mark one of these combinations during the same decision cycle.
- Terminated Goals are erased by NGS after a certain period of inactivity. If you do not want your terminated Goals to be erased, you should change the settings in your version of settings.tcl.

Advanced: Since the underlying implementation of Goals in NGS operates at the more general level of "considerables", you can create your own set of considerables through the use of the Tcl procedure **`NGS_add_considerable_item_type`**. Each type of considerable gets its own set of activation pools, which always hang in a fixed place from the top state.

3.1.1 Goal Justification Support

NGS does not commit a developer to using one type of support (i-support or o-support) for object creation, and there is no hard-and-fast rule for when to use o-support over i-support. I-supported objects go away when their creation condition no longer matches, with goal cleanup being handled by Soar's typical truth maintenance system. On the other hand, O-support is preferred if you need to keep Goals around after they're terminated (e.g., for reasoning about an agent's history). For O-Supported Goals, the NGS infrastructure will remove terminated goals after a fixed amount of time (see considerable-management.soar for the timeout.)

3.2 Tags

Tags are used to mark objects with process-related information (e.g., the activation state of the objects) that does not contribute to the basic definition of the object. For instance, the deadline for when an object expires, or the time at which some event should be triggered, or the whether a step in a multi-step process is complete, are often treated as tags. This is an organizing principle to keep the object definition from becoming too cluttered. When in doubt as to whether to add an annotation to an object as a tag or directly to the object body, you should ask yourself if the annotation is an intrinsic part of the object – do you see this annotation as a key part of that object? If the answer is no, then you should instead put the annotation in the tags structure.

When you use an **NGS_create-*** RHS macro (**NGS_create-operator, NGS_create-goal**, etc.), NGS automatically add the **^tags** attribute to the created object.

The LHS macros **NGS_is-tagged** and **NGS_is-not-tagged** can be used to test for the existence of any sort of tag on an object.

For more specific built-in NGS tests on a goal's activation state, you can use these LHS macros: **NGS_is-active, NGS_is-not-active, NGS_is-suspended, NGS_is-not-suspended, NGS_is-achieved, NGS_is-not-achieved, NGS_is-unachievable,** and **NGS_is-not-unachievable**.

To manipulate a tag on an object, use the **NGS_tag** RHS macro.

To mark activation state on Goals, use these RHS macros: **NGS_mark-active, NGS_mark-suspended, NGS_mark-achieved,** and **NGS_mark-unachievable**. Each of these has a corresponding **NGS_unmark-*** RHS macro.

3.3 Object Typing

As mentioned earlier, NGS's object typing scheme is an attempt to represent object/class representation in Soar. An object's type information is stored in the **^type-info** structure on each object. The **NGS_create-*** RHS macros (**NGS_create-operator, NGS_create-goal,** etc.) automatically add the **type-info** attribute to the created object. This **type-info** structure can be accessed for testing using the **NGS_is-type, NGS_is-not-type, NGS_is-most-derived-type, and NGS_is-not-most-derived-type** LHS macros.

When being passed to a macro, an object's types must always be a list of strings. NGS considers this list to be an ordered type hierarchy, with the first item in the list being the most general type, and the last item the most derived type. As an example, if you wanted to create a "watch-resource" goal with a "fuel-monitor" subtype, you could do it like this:

```
[NGS_create-goal <goals> {watch-resource fuel-monitor}]
```

Matching to a goal's specific type on a LHS is implemented using the macro NGS_is-most-derived-type. Here's an example:

```
[NGS_is-most-derived-type <goalname> typename]
```

4 Using NGS in a Soar Agent

4.1 Loading NGS

In order to use NGS within a Soar agent, you must load the NGS source code.
Do so by sourcing the file "NewGoalSystem.tcl" in the NGS root directory.

All of NGS' code will be contained within the "NGS" tcl namespace, but all of the macros listed in "documentation/quick-ref.txt" are exported for your convenience.

Thus, if you want to call NGS_mark-active, you may simply use it, e.g.:

```
[NGS_mark-active <foo>]
```

But if you wish to reference NGS' $active_goals variable, you will need to precede it with a TCL namespace, e.g.:

```
$NGS::active_goals
```

4.1.1 "Enhanced-load" automatic file loading infrastructure

NGS provides some built-in macros to help in the loading of a tree of Soar files. These macros may be found in the "Enhanced Load" file loading infrastructure. This infrastructure is contained in the file enhanced-load.tcl. It's designed to aid in the sourcing of tcl files, Soar files, and subdirectories.

The most important parts of enhanced loading are the "NGS_echo-source" command, and the "NGS_load-soar-dir" command.

The "NGS_echo-source" command is like Soar's built-in "source", but it also prints out some information about the file it's loading to the agent console – this helps with debugging the load progress. This printing is disabled if the variable NGS_NO_DEBUG_PRINTING has been set to yes (see the section on configuring settings, below).

The "NGS_load-soar-dir" macro is used to automatically switch into a directory and load the "load.soar" file in that directory.

For examples of how to use Enhanced Load, see NGS' file ..\load.soar.

4.2 NGS meta-management

NGS has a built-in mechanism for determining the settings of the NGS base code, and the code for each module. The NGS base comes with a set of default settings (found in default_settings.tcl). These defaults are loaded upon NGS' loading. In case those settings do not suit your project's needs, there is also an included override mechanism. If you wish to use this, you should create your own settings file, and set the tcl variable `NGS::NGS_settings` to the name and path of your custom settings file. This path should be relative to the location of the "default_settings.tcl" file.

An example of how to use the settings override…. Create your settings file:

```
echo "NGS_create-soar-var NGS_NO_DEBUG_PRINTING yes >> my_settings.tcl
```

Then put this in your load.soar file, *before* you call NewGoalSystem.tcl:

```
namespace eval NGS {set NGS_settings ../my_settings.tcl}
```

This custom settings file will be loaded after the default settings, allowing you to change or add to the variables created by the default settings.

4.2.1 Tcl "shortcut" variables

Where it makes sense, we have defined Tcl variables for abbreviating paths to common object locations, or for object locations that need some flexibility (eg, for future change).

This is a recommended method that can save time and effort, instead of attempting to remember the names of the attributes along the five-step path to a particular data area.

See standard-variables.tcl for example definitions, and look to the code for usage.

4.2.2 Source Control Variables

The sourcing of each optional module included with NGS is dependent upon a module-specific control variable. If the variable is defined (set), the module will be loaded. If not, the module will not be included in an agent's source. See default_settings.tcl for these variables - and override them in your custom settings file as needed. Note that no modules ship with the current release of NGS.

4.2.3 (Tcl Macro) LHS Fragments

LHS Fragments are LHS matching macros that generate LHS matches and bindings - they're used to shorten Soar productions and make them more consistent and easier to read by generating portions of the left-hand-side of a Soar production. See infrastructure/templates/lhs-fragments.tcl.

4.2.4 (Tcl Macro) Manipulators

Manipulators refer to RHS object generation or modification code. They include Tcl fragment generators for adding information to an existing object, as well as the constructors for goals and other objects. As you create new objects, you should create constructors (new Tcl macros) for them as well. Manipulators and constructors should always be used on the right-hand-side of a production, and will likely cause load-time failure if used on the left-hand-side of a production.

4.2.5 Tcl macros – notes on expansion of macros

NGS makes extensive use of Tcl as a macro language for creating Soar productions or production sub-pieces. Any production using NGS code – especially Tcl variables or macros – must be quoted ("..."), rather than using curly-braces ({....}). This is because Tcl will treat anything in curly braces literally, when instead NGS relies on expansion of its macros which only happens when in quotes.

Note that different versions of Soar may support Tcl differently.

If you want to see what a production will look like when it is fully expanded, you can temporarily replace the 'sp' at the beginning of the production with an 'echo'. This will cause the tcl interpreter to output your production as text to the screen instead of loading it into Soar memory. The Soar IDE (Eclipse plug-in) at some point supported showing the non-processed Tcl version of a rule. See http://code.google.com/p/soaride/

4.3 Debugging

By default, status of operators and goals are not output to the screen. These can be turned on/off using interface switches (see settings.tcl).

5 Developing with NGS

5.1 NGS Conventions

5.1.1 NGS baseline Coding Conventions

All the NGS Tcl templates are prefixed by "NGS_". This should prevent clobbering of other procedures. All the NGS productions are prefixed by "ngs*" to prevent clobbering of other productions.

5.1.2 NGS-derived Production Naming

Wherever possible NGS-based code should attempt to follow these naming conventions:

```
<context>*propose*<operator-name>*<details>
<object>*elaborate*<details>
<operator>*apply*<details>
<operator>*apply*create-goal*<goal-name>*<details>
<supergoal>*create-<sub>goal*<goal-name>*<details>
```

By placing the context/object/operator first, when printing productions in soar, all productions with the same context will appear together.

Where context might be a goal name, a problem space, an operator, etc., <details> might give extra conditions under which the production would fire, or different cases, etc.

5.1.3 Naming of NGS Files and Directories

When altering or extending NGS:

1) Create a new file for any well-encapsulated process or module
2) Create a new directory when there is a process that requires multiple related files to be contained.
3) Name files and directories with regard to the processes they address.

Typically with directories, there is a load.soar that's used to source all files inside that directory. E.g.,

```
common/
    load.soar
    my-behaviors.soar
    my-elaborations.soar
```

5.1.4 Documentation

The documentation directory stores NGS documentation, including this document, and other API documents. For code documentation, NGS itself uses inline documentation in the following style:

```
##!
# @production <name>
# @brief <brief-description>
#
# @desc <longer description>
#
# @type <elaboration/application/proposal/etc>
# @operator <where appropriate>
```

5.1.5 Unit Testing

Goals (and other considerables) should be exercised by at least one (if not more) unit tests. See testing/*.soar for example unit tests.

5.2 Behavior Design Patterns

First and foremost, NGS is an implementation of Forest of Goals pattern, and uses Declarative Goals. See "Behavior Design Patterns" by Taylor and Wray (2004) for examples of these.

5.3 NGS History

The current version of NGS is the result of work by Jacob Crossman, Glenn Taylor, Sean Lisse, and Bob Marinier at SoarTech. It has gone through multiple iterations, with more and less functionality. The current version represents a minimal set of functionality to help users start building Soar agents using NGS. We may add extensions or new modules in the future.