The O2S Goal-oriented Scripting Engine:
preliminary architecture and usage specification

Author: Justin Mazzola Paluska
Contact: jmp@mit.edu
Version: 3841

Contents

1 Overview

This document describes the architecture of a Goal-oriented scripting engine. It follows a general model of Goal-oriented planning designed for easy implementation of adaptive behavior inspired by the needs of the pervasive computing community. While designed in parallel with O2S nPops, the scripting engine is separate from any underlying component model, and hence potentially applicable to a wider range of tasks than just those of pervasive computing.

Primary inputs to the engine are *Goals*, each a parameterized representation of a desired condition. The output is arbitrary scripted behavior—typically the dynamic assembly of a system of software and hardware components into an application or subsystem that implements the desired condition. The engine itself is completely naïve with regard to specific Goals and ways of satisfying them; it depends on a universe of scripted *Techniques*, each an alternative means to achieve some specified Goal. Each Technique may specify sub-goals that must be satisfied as a prerequisite to its application; it thus may constitute a formula for decomposing some high-level Goal into sub-goals plus some scripted behavior. This leads to a conceptual *goal tree* of goals, techniques, and sub-goals generated during the planning process carried out by the scripting engine, reminiscent of those generated by Prolog and similar logic languages. The selection of specific techniques to satisfy a particular goal is a heuristic choice made by the engine, and may reflect availability of appropriate resources and other environmental variables. The selected subtree (or *plan*) may then be executed; during execution, the engine is able to re-evaluate its choices, select alternative plans, and revise the chosen implementation strategy.

Although the basic architecture of the goal-oriented scripting mechanism is both platform- and language-neutral, the current implementation is based on (and implemented in) Python. Python thus provides the syntactic and semantic substrate for its scripted Techniques, and is used here consistently for API specifications and concrete examples.

1.1 Programming in the O2S Goal model

Our model provides a way of structuring applications so that the code responsible for any piece of the application may be evaluated and replaced as the environment varies. Like traditional programming models that separate implementations from interfaces, programmers write *Techniques* that satisfy the specifications of *Goals*. Unlike traditional models, the application programmer does not choose which specific Techniques will be used at runtime. Instead, goal-oriented programming maintains the indirection between Goals and Techniques and offloads the choice of which Techniques will satisfy which Goals to a runtime *Planner*.

The Planner's job is to choose among the Techniques that satisfy the programmer's Goals. Since there may be more than one Technique that can satisfy the programmer's Goal, Techniques include a mix of code meant to help the Planner evaluate the quality of the solution the Technique can provide as well as code to actually carry out the Goal. Techniques are also Goal-oriented and can declare sub-goals that the Planner will satisfy much like the application's high-level Goals.

1.1.1 Main Concepts in O2S Goal-oriented Programming

The O2S model of goal-oriented programming centers around four main concepts: Goals, Techniques, Solutions, and the Planner:

Goals and Goal Instances A Goal is a parameterized specification of a class of high-level applications intents or desires. Syntactically, it is a disembodied generic procedure with typed (positional and keyword) parameters. A Goal instantiated with a particular set of parameters is formalized as a Goal Instance, whose syntax is that of a procedure call.

Goals specify the interfaces and exposed state that a programmer should expect from the Techniques the Planner chooses.

Technique and Technique Instances A Technique is a parameterized template for satisfying a Goal. Goal Instances are satisfied by Technique Instances, which are live, running objects with their parameters bound to the parameters of the Goal Instance. Techniques store their state in *solution* objects that users of the Planner can inspect and use in their application.

Solutions Associated with each Technique Instance and Goal Instance is a Solution. Solutions hold named values called Properties. Properties are used by the Planner and are available to the application for evaluating Technique Instances.

An important Property of a Solution is the `satisfaction` property. `satisfaction` is a floating point number between 0 and 1. For a Technique Instance, the `satisfaction` represents how well that particular instance can satisfy its Goal Instance. 1 is perfect and 0 is unsatisfied.

The Solutions of Goal Instances are copies of the Solution of the Technique Instance satisfying that Goal Instance with the highest `satisfaction` value.

Planner The Planner is the scripting engine behind Goal-oriented programming. It searches for, evaluates, chooses among, and executes Techniques in order to satisfy an application's high-level goals. This process is called *planning*.

For each high-level Goal Instance, the Planner maintains a data structure called a Plan. The Plan is a tree structure of alternating layers of Goal Instances and Technique Instances that represents all of the possible implementation choices of the application.

1.1.2 The remind application as a Goal-oriented program

We illustrate goal-oriented programming through a "meeting minder" application called `remind` as the primary example. `remind` takes in a time to send the reminder and a text message with the actual reminder. The source code for `remind` is available from the O2S repository in the `o2s/planner/docs/examples` directory.

Since `remind` must notify a user of meetings, `remind`'s main high level goal is to send text to a user. When it is time to notify the user, it asks the Planner to satisfy a `SendText` high-level goal. The Planner responds by finding all of the Techniques that satisfy the `SendText` goal and recursively finding Techniques to satisfy their sub-goals. A typical set of Techniques will be ones to display dialog boxes or send text messages to the user's cell phone. The Techniques with the highest probability of reaching the user are rated with higher satisfaction values. Finally, the Planner chooses a set of Techniques rooted at the `SendText` Technique with the highest satisfaction to implement and runs them.

1.2 Tutorial Trails

The rest of this document is split into sections on:

- Using the Planner from the Command Line,
- Using the Planner in an Application, and
- Programming the Planner.

2 Using the Planner from the Command Line

The Planner can be called from the command line using the `planner.py` script. The `planner.py` script creates a Plan from a goal specified on the command line, evaluates it until it finds a viable Plan, and finally executes the Plan. It prints out a snapshot of the of the top-level Plan before cleaning up the entire process and exiting.

For example, we can invoke the `SendText` goal that `remind` will use with the following command:

```
python planner.py SendText -S user=$USER -S message="You're late!"
```

The behavior of the command will vary depending on what Techniques are installed and what needs to be done.

2.1 Exploring the Planning Process

In order to get a better handle on what the planner is doing, you can pass it the `--inspector` command-line options with the name of an Plan Inspector package. For example, running the same command with the console inspector leads to the following output:

```
python planner.py --inspector=console_viz \
   SendText -S user=$USER -S message="You're late!"

(Discovering for 10.00 seconds. Please be patient.
REGISTRY LIST:  []
Plan has stabilized:
Goal: SendText(message=You're late!,user=jmp) Snapshot({'satisfaction': 0.125}) Marked=True
|  Terminal Snapshot({'satisfaction': 0.01}) Marked=False
|  Zephyr Snapshot({}) Marked=False
|  PopUp Snapshot({'satisfaction': 0.125}) Marked=True
****************************** Executing Plan *******************************
Executed Plan:
Goal: SendText(message=You're late!,user=jmp) Snapshot({'satisfaction': 0.125}) Marked=True
|  Terminal Snapshot({'satisfaction': 0.01}) Marked=False
|  Zephyr Snapshot({}) Marked=False
|  PopUp Snapshot({'satisfaction': 0.125}) Marked=True
```

Here, there are three Techniques that can `SendText`. The `PopUp` Technique has the highest satisfaction, so it gets marked for execution when the Plan stabilizes and later is executed. The other Techniques are not chosen because they have lower (or non-existent satisfaction values). In addition to the console inspector, there are also 2-D (the `plannertree` option) and 3-D tree (`plannertree3d`) inspectors as shown below. If these visualizations are not enough, Inspectors are pluggable: see the Plan Inspectors section for more information.

2.2 Command-line Arguments and Environment Variables

The general syntax for the `planner.py` command is:

```
planner.py [planner arguments] goal [goal arguments]
```

The goal arguments define the parameters to the top-level goal. Goal arguments are keyword-arguments, so values passed to each goal argument take the form `name=value`. For example, `-I max=100` will set the `max` goal argument to the integer 100. You can also define string arguments using `-S` as seen in the `SendText` examples above.

The Planner can also optionally take in a Technique search path. This defaults to the current directory, but can either be set with the environment variable `TDB_PATH` or the command-line argument `--tdb-path`. To set more than one search directory, delimit the paths with colons in the environment variable or invoke the command-line argument multiple times. For example:

```
export TDB_PATH=/usr/lib/teqs:/home/user/teqs
```

is the same as using `--tdb-path=/usr/lib/teqs --tdb-path=/home/user/teqs` on the command-line.

The full help for the Planner command-line utility is available by running the Planner with the `--help` option.

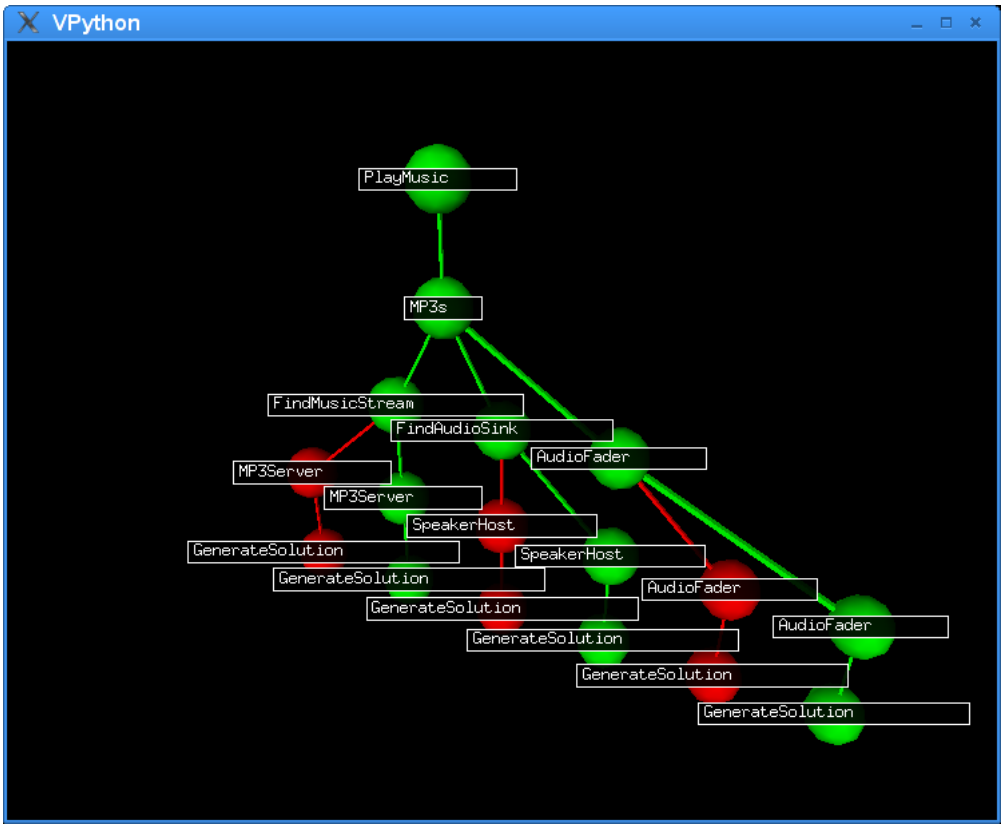

Figure 1: The `plannertree3d` inspector while the Planner is playing MP3s to a Speaker.

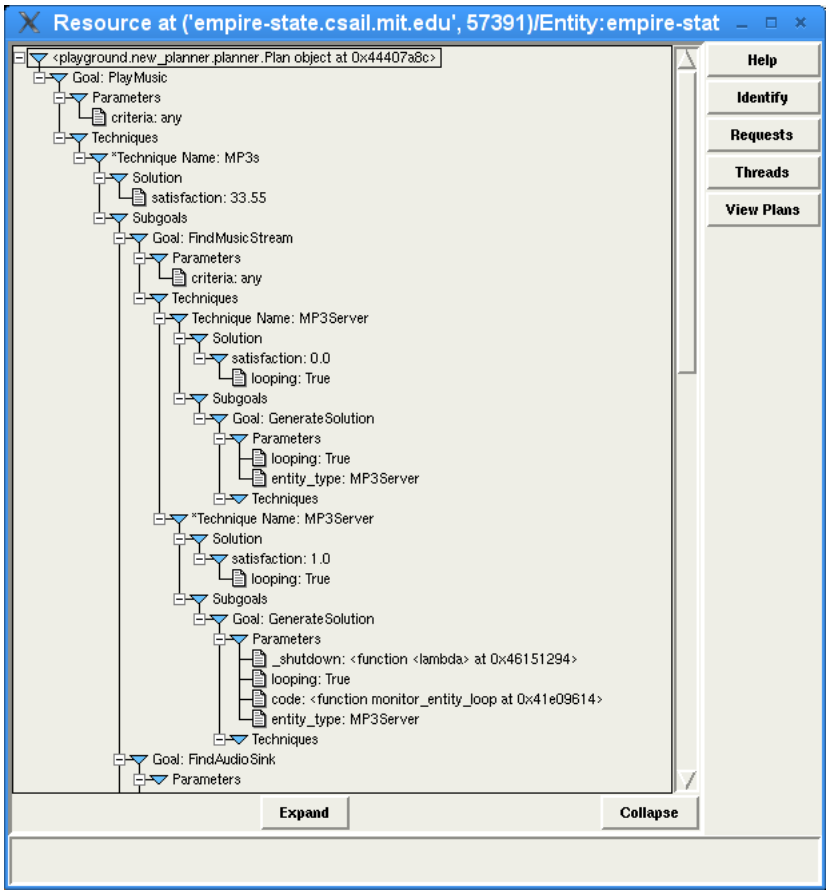

Figure 2: The `plannertree` inspector for the same plan.

3 Using the Planner in an Application

In order to tell the Planner to plan a specific Goal, an application must:

1. Create a new `Goal` object that represents the application's high-level intent.
2. Ask the Planner to plan the Goal. This returns a new `Plan` object that allows the application programmer to control aspects of the planning process.
3. Ask the Planner to evaluate and execute the Plan.

The `remind` application does exactly this with a `SendText` goal when it determines that it is time to remind the user. Therefore, the outline of the `remind` application boils down to:

```
import playground.new_planner.planner as planner
import playground.new_planner.techniques as techniques

# other necessary imports

if __name__ == '__main__':

    # parse the command line to determine how long to sleep for
    # before sending out the reminder as well as the actual reminder
    # text.  In particular, set the sleep_time and the message
    # variables.

    # wait to do the actual reminding
    if sleep_time > 0:
        time.sleep(sleep_time)

    # finally, call the planner
    plannerObj = planner.Planner()
    goal = techniques.Goal('SendText',
                           user=o2s_globals.O2S_USER,
                           message=message)
    plan = plannerObj.plan(goal)
    plan.execute()
```

In more detail:

`plannerObj = planner.Planner()` Creates a Planner instance using the default Technique Database. See Command-line Arguments and Environment Variables for more information about the Techniques that comprise the default Technique Database.

The Planner can take additional arguments such as a different Technique Database. See Planner API for more information.

`goal = techniques.Goal('SendText', ...)` Creates a Goal Instance to be fed into the Planner.

`plan = plannerObj.plan(goal)` Register the `SendText` goal with the Planner and save away a handle to the Plan object underlying that Goal.

`plan.execute()` Run enough evaluation to find a set of Techniques that satisfies the `SendText` goal and implement the code that they embody. There are a variety of ways to control evaluation and execution of the Plan; `execute` is the simplest method possible. See Plan API for more information.

Most applications can probably use the Planner with just this much code. In fact, the code that calls the Planner in `planner.py` doesn't do much more than this.

3.1 Planner API

Planner objects are fairly simple, as they just serve as "Plan Factories" and as the initializer for PlanInspector objects. They have the following API:

`__init__(TechniqueDatabase:tdb)` Create a new Planner Object backed by `tdb`.

`plan(Goal:g)` **returns** `Plan` Register `g` as a new high-level application intent. Return a `Plan` object for `g`. See the Plan API for more information.

`add_inspector(PlanInspector:inspector)` Add `inspector` to the set of inspectors inspecting this Planner.

3.2 Goal API

Goal objects are immutable representations of high-level goals. Goal Instances are created by:

`__init__(name, **kwargs)` Where `name` is the name of the Goal and `kwargs` is a list of Goal parameters.

3.3 Plan API

A Plan is a handle to the planning process for a single top-level goal.

Plan objects offer both a high-level API and a more primitive API out of which the high-level API is constructed. The high-level API has three functions—evaluate, execute, and update—corresponding to the main functions of the Planner.

The five primitive operations are single-step evaluate, execute, update_to, shutdown, and cleanup. Single-step evaluate chooses a one evaluation phase of one Technique to run and update Technique's solution object. Execute chooses the best Plan given the evaluation so far—if there is one—and executes it. Once a plan is executing, the application may continue to call single-step evaluation and can update_to a new set of Techniques if the Planner finds them.

To garbage collect Plans and their associated resources, Plan objects have two cleanup operations: shutdown and cleanup. Shutdown is the opposite of execute--it takes down a running Plan. The Plan can be restarted by running execute on the plan. Cleanup stops all evaluation on the Plan. The only way restart a cleaned-up plan is to ask the Planner to `plan()` the goal again.

Finally, the Planner offers an introspection API through which the state of individual solutions are accessible.

3.3.1 High-level API

`evaluate()` **returns** `SnapshotNode` Repeatedly run single-step evaluate until there's nothing more to do and return a snapshot of the current state of the tree.

`execute()` **returns** `SnapshotNode` **If the Plan can be** executed, execute it. Otherwise, continue evaluating until it can be executed.

3.3.2 Primitive Plan API

`single_evaluation()` **raises** `StopIteration` Run a single stage of evaluation. Raise StopIteration when there is no more evaluation to do now.

`pending_evaluation()` **returns** `bool` Return True if more evaluation can be done.

`wait_evaluation(timeout)` Block for timeout or until some evaluation can be done, whichever comes first.

`raw_execute()` **raises** `ExecutionFailure` Execute the top-level goal. Raise ExecutionFailure if the Plan cannot be executed.

`shutdown()` Stop an executing Plan if it is executing.

`cleanup()` Prepare a Plan for garbage collection by calling `shutdown` (if necessary) and stopping all evaluation.

3.3.3 Introspection API

`get_snapshot()` **returns** `SnapshotNode` Return a snapshot of the Plan.

3.4 SnapshotNode API

A SnapshotNode is the base class for introspection nodes. SnapshotNodes are typically produced by the `Plan.get_snapshot` method or the execute/upgrade methods of `Plan`. Every SnapshotNode represents some Goal or Technique in the Plan.

A SnapshotNode exports several attributes:

`parent` The parent of the SnapshotNode, or `None` if the SnapshotNode is the root of the Plan.

`solution` An immutable view of the backing node's solution at the time the SnapshotNode was created.

`marked` True when this node is executing or chosen for execution.

as well as a few methods:

`is_different(other)` **returns** `bool` Returns True if set of marked nodes in the sub-tree rooted at the SnapshotNode is structurally different than the marked nodes rooted at `other`.

`is_failed` **returns** `bool` Returns True if the implementation represented by this SnapshotNode has failed.

`get_children` **returns** `Iterable` Return an iterable over the SnapshotNode's children SnapshotNodes.

4 Programming the Planner

Programmers add new Goals into the O2S Planning System by writing Techniques. Techniques are written as a sequence of different stages. There are two main types of stages, evaluation stages and execution stages. Evaluation stages help the Planner determine what the requirements of the Technique are as well as how well the instantiated Technique can satisfy its goal instance. Execution stages contain the script code to actually satisfy the Goal.

Techniques can be embedded in Python scripts and may call arbitrary Python code.

4.1 Example Technique

A sample technique looks something like the following:

```
to Fact(n):

    via Recursion:

    first:
        # We test whether this techique applies to our parameters:
        if goal.n <= 0:
```

```
            planner.fail("Cannot take factorial of non-positive number!")
        else:
            solution.satisfaction = 0.5

    subgoals:
        n1 = Fact(n=goal.n-1)

    eval:
        # set satisfaction based on the expense of the subgoal
        if subgoals.n1.satisfaction > 0:
            sat = .90 * subgoals.n1.satisfaction
            solution.satisfaction = sat
        else:
            solution.satisfaction = 0.0

    exec:
        # This is called AFTER the exec phase of each subgoal has completed.
        # Our job is to fill in our value:
        solution.value = goal.n* subgoals.n1.value
        solution.satisfaction = 0.5
```

Like Python files, Techniques use indentation to define structure and `#` to mark the beginning of comments. The first line:

```
to Fact(n):
```

declares that this Technique solves the `Fact` Goal of the single argument `n`. The next line:

```
via Recursion:
```

names the Technique, in this case `Recursion`. The rest of the Technique source define a series of stages. Stages take the form:

```
stage-type:
    ...contents...
```

The `stage-type` must be indented to the same level as the `via` statement. Each of the stage types and what forms their contents is outlined below. Both "evaluation" stages and "execution" stages take this form. Except for a few instances noted below, evaluation stages can be appear in any order and may repeat types—for example, to declare multiple subgoals. Execution stages may not be repeated.

4.2 Evaluation Stages

The planner will try to gradually build up execution requirements and evaluation, often by re-running evaluation stages as the environment changes. As a consequence, evaluation stages must be idempotent. To aid the programmer, solution objects keep track of which assignments are made in which stages and will restore the state of the solution to the state it was previously in when the stage was run.

To quicken the pace of evaluation, the planner may also cut off phases that run for too long[1] .

The types of evaluation stages are:

first perform basic initialization, akin to a constructor in an object system. The `first` stage must be the first stage in the Technique if it exists. This stage contains Python code.

[1] The current planner implementation does implement this cut-off.

subgoals declare subgoals. This stage is a list of declarations of the form:

```
[subgoal] SUBGOAL_NAME = SUBGOAL(...SUBGOAL_ARGS...)
```

In later stages, the solution object associated with `SUBGOAL_NAME` is available as `subgoals.SUBGOAL_NAME`.

eval perform an evaluation function. This stage contains Python code.

foreground This stage functions exactly like the eval stage except that the planner will not cut it off for running too long. However, the Planner runs the code in in such a way to block the Technique, but not the Planner as a whole. (For example, it might use a thread for this stage.)

The Planner will move to the next evaluation stage only when the current stage has successfully executed. For most stages, "successful" execution means that stage did not call the `fail` method. For subgoal stages, "successful" execution means that the requested subgoals have not failed.

4.3 Execution Stages

The next two stages actually build—versus evaluate—an implementation:

exec commit this technique and start execution of its implementation. This stage is required. There may only be one exec and it must be the first execution stage.

shutdown release any locked resources and clean up. This must be the last stage if it exists.

The exec stage is executed only if (1) all of the previous stages have executed successfully and (2) the Planner chooses the Technique for execution. If the Technique's execute phase does not complete successfully or either of the two conditions above no longer hold the cleanup stage of the Technique will be executed. The Planner may re-run evaluation stages while the implementation is running in order to determine if the execution environment has changed so much as to require the Technique be shutdown.

If a Technique can handle hot swapping of the Techniques supporting its subgoals, the Technique can also specify update stages. The signature of an update phase is more complicated than normal stages since the Planner must be told what subgoal can be swapped:

```
update <sg_name> from <old variable>
```

`<sg_name>` is the name of the subgoal that can be updated.

`<old variable>` is a variable name that will hold a handle to the old Technique.

4.4 Environment

The Technique programmer has access to four state variables:

`goal` the Goal Instance the Technique is aiming to implement. The goal arguments are accessible as attributes.

`solution` a Solution object for that Technique. `solution` serves two purposes. First, it is the method of communication between stages. Any state that must be saved by a Technique for use in later stages should be saved in a Solution. Second, it exports public Properties of the Technique that will be accessed by parent Techniques. These exported Properties can be used in evaluation in Techniques higher in the Plan.

Properties of `solution` are accessed as attributes. Furthermore, all Properties are considered to be public properties, unless the property name starts with `_`. For example:

```
solution.satisfaction = .5
```

sets the public property `satisfaction` to .5, while:

```
solution._var = 42
```

makes `_var` to be a private property and sets its value to 42. `_var` will be accessible only to the Technique that declared it and not to other Techniques.

`subgoals` object with every subgoal's solution as an attribute. A subgoal with declared with name `sg` is accessed through `subgoals.sg`.

`planner` access to utility methods in the Planner itself, mainly to signal to the planner a change in state.

Techniques should not raise Exceptions in evaluation stages. If they do, the current implementation just pretends that the Technique called `planner.fail`. In the `exec` stage, Techniques should raise `ExecutionFailure` if they cannot execute for any reason.

4.4.1 Methods of the `planner` Object

Techniques are provided with the following methods through the `planner` variable:

`planner.fail(reason)` Mark that this Technique has failed, with an optional reason. This halts execution of the Technique until something in its previous stages changes.

`planner.quit(reason)` Signal to the Planner, from within the Technique, that the current Plan is done executing and can quit. Only executing Techniques can successfully call `quit`.

`fail` is different from `quit` in that the Planner will try to plan around any failed Techniques, but will shut down when told to `quit`.

4.4.2 Technique Node Cloning

Sometimes one Technique may return a multitude of results. For example, a Technique that searches its environment may find many resources, files, or pages that match the search criterion. In order to push more decision making from Techniques to the Planner, the Planner will try to clone Techniques whenever they produce more than one solution. This way, each solution is associated with a single Technique and the Planner can choose among them when satisfying higher-level goals.

Techniques themselves cannot have more than one solution, however, they can request execution of code that may generate multiple solutions though the GenerateSolution subgoal. Whenever the code makes a new solution `sol`, the Technique that executed the code will get cloned. The cloned is exactly the same as the original Technique except that the solution bound to the GenerateSolution will be `sol`. If the arguments to the GenerateSolution subgoal change for some reason, then all of the clones will be destroyed, the code shutdown, and the process started over again.

4.5 Example SendText Techniques

For example, consider a few different techniques for sending a text string to a user. The simplest possible Technique is just one that prints out the message to the user's terminal:

```
to SendText(user,message):

    via Terminal:

    first:

        solution.satisfaction = .01

    exec:

        print goal.message
```

At the least, a Technique should set its satisfaction value and have a non-empty `exec` stage, so the `SendText via Terminal` technique is the simplest Technique possible. More example techniques are in the `send-text.teq` file of the remind.py application.

4.6 Goal Library

In addition to user-defined Goals, the planner provides a Goal Library to be used to extend the functionality of Techniques.

4.6.1 GenerateSolution

`GenerateSolution(code=, _shutdown=, *callable_args, **callable_kwargs)`

Run the callable specified by the code argument and return one of the solutions that the code generates as the subgoal's solution. Note that if the code produces more than one solution, the Planner will try clone the calling Technique once for each solution produced. See Technique Node Cloning for more information.

4.6.2 Arguments

The code argument will be called in the following way:

```
code(sf, *callable_args, **callable_kwargs)
```

where *sf* is a SolutionFactory object that can use used to make solutions. It is called in its own thread.

The _shutdown argument specifies a way to stop the code from running if it still running when the Goal is no longer needed. _shutdown will be called, when the time comes, as:

```
_shutdown()
```

The code will be shutdown and restarted if any of the callable_args or callback_kwargs lists change, i.e., when the subgoal phase is restarted.

4.6.3 Idioms for the code argument

The GenerateSolution goal can be used in order to monitor something in the background as well as generate a set of solutions. In both examples, the Technique has the following subgoals phase:

```
subgoals:
  soln = GenerateSolution(fn, ...)
```

The major difference between the two nodes is in how the `fn` uses the SolutionFactory that the planner passes it.

In either case, the `fn` writer must keep in mind two details. First, whenever the code wants to notify the Planner of a change in a solution, it must call the solution's `notify()` method. Second, each solution that the SolutionFactory generates will be interpreted literally by the planner, so if the solution is valid, it must have a `satisfaction` attribute with a value greater than 0.

- Monitoring

  Here, `fn` will make only one solution from the SolutionFactory.

```
def fn(_sf, ...):

  soln = _sf.new_solution()

  while monitoring:
```

```
        if something_changed:
            soln.attr = whatever
            soln.satisfaction = new_satisfaction
            soln.notify()
```

- Multiple Solution Generation

  Here, `fn` will make multiple solutions as it finds things in the environment. Usually it is a good idea to keep a dictionary mapping each value and solution. This way, if a value is no longer valid, its solution can be set to 0.

```
def fn(_sf, ...):

    solns = {}

    while running:
        if found_new_thing:
            soln = _sf.new_solution()
            solns[new_thing] = soln
            # modify the attributes of soln
            soln.notify()
        elif something_changed:
            soln = solns[changed_thing]
            # ... modify soln
            soln.notify()
        elif thing_now_invalid:
            soln = solns[invalid_thing]
            soln.satsifaction = 0.0
            soln.notify()
```

5 Extending the Planner

5.1 Plan Inspectors

In addition to the built-in inspectors, the Planner can use any object that inherits from the `PlanInspector` class as an inspector and visualization tool. A `PlanInspector` consists of a set of callbacks for specific events in the evaluation and execution of a Plan. Each callback is supplied with the *Plan* object that should be inspected by calling `get_snapshot`.

The methods in this `PlanInspector` base class do nothing, so sub-classes may only implement the callbacks they wish.

5.1.1 Plan Inspector API

`__init__(planner)`: Create a new PlanInspector for `planner`.

`new_plan(plan)` Called after `plan` is initially created—this will be the first callback for the passed in `plan` object.

`stable_eval_cb(plan)` Called when there is no more evaluation to do for `plan`‘.

`before_exec_cb(plan)` Called before `plan` is executed.

`after_exec_cb(plan)` Called after `plan` is executed.

`before_update_cb(plan)` Called before `plan` is updated.

`after_update_cb(plan)` Called after `plan` is updated.

`before_shutdown_cb(plan)` Called before `plan` is shutdown.

`after_shutdown_cb(plan)` Called after `plan` is shutdown.

`cleanup_cb(plan)` Called when `plan` is cleaned up. `cleanup_cb` is the terminal call the Planner makes for `plan`, so the PlanInspector can clean up state related to `plan` during this call or any time after it.