aLIGO Guardian overview

Jameson Graef Rollins

January 7, 2014

Introduction

Guardian is the new aLIGO automation system.

It will take over for all “scripts” and old-style auto-lockers to handle automation of the interferometers and all subsystems.

Overview

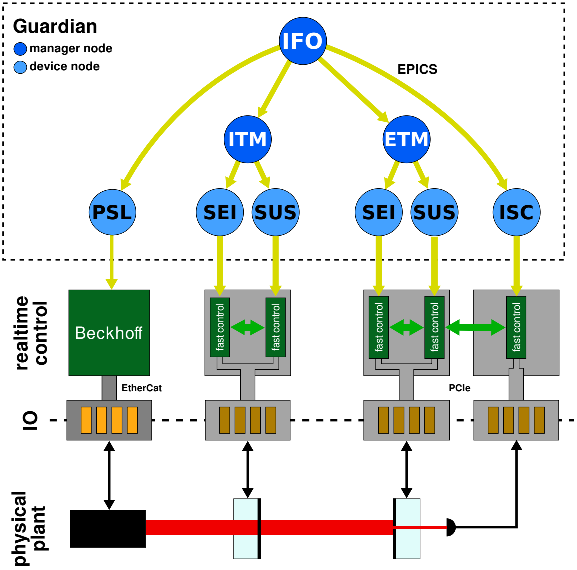
The implementation has been much improved, but the basic concept remains the same:

- Distributed guardian processes (**nodes**) oversee particular domains of the interferometer.
- Each node understands **states** for their domain. State code describes transitions and verification of the state of the domain.
- A hierarchy of nodes control the full IFO, with top level **manager** nodes controlling sets of lower-levels nodes, down to lowest level **device** layer that talks directly to the real-time front-ends and Beckhoff.

Overview

Top level manager nodes control lower level subordinates, down to device nodes that talks directly to the real-time system.

A single IFO manager will sit at the top, accepting state requests for the entire interferometer.



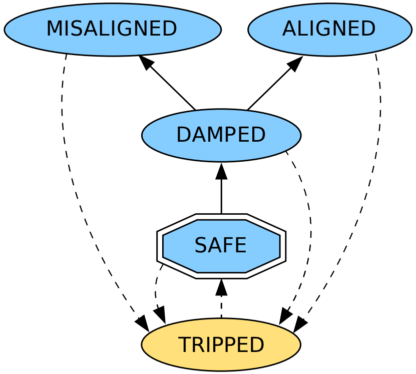
System and state behavior

System behavior

Each node is programmed as a **state machine**.

The state machine can be represented as a **directed graph**, where states are connected by **edges**.

Each edge represents a possible transition between states.

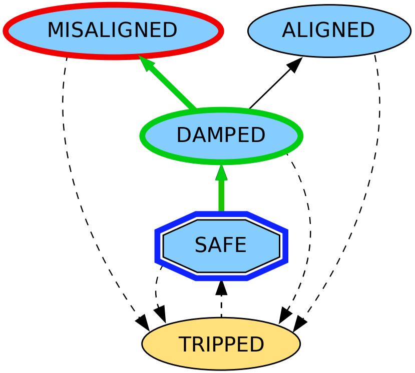


System behavior

Guardian accepts commands in the form of a **request state**.

Guardian then looks at the **current state** and calculates the shortest **path** to reach the request.

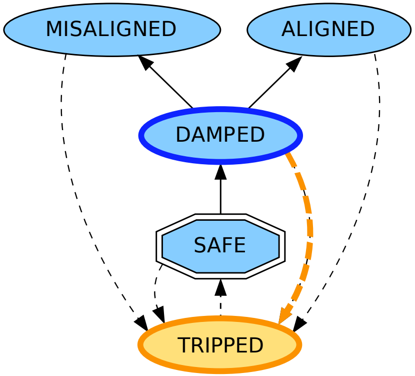
The code for the current state is executed to completion. Once done, Guardian transitions to the next state in the path and executes its code. Once it reaches the requested state it holds there.



System behavior

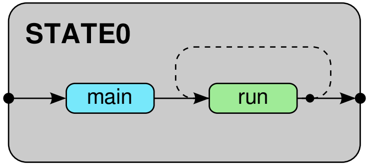
States can also specify **jump** transitions that bypass the normal dynamics of the graph.

Jump transitions are used for recovery from conditions that are in some sense undesirable (e.g. lock loss, watchdog trip, etc).



State behavior

The states themselves have very simple behavior. There are two state *methods* (i.e. functions):



main executed **once** immediately upon entering state. This is the primary state code.

run executed **in a loop** continuously until it returns True or a jump state name. This should be used to check conditions for completing the state and/or jumping to a recovery state.

Code structure and syntax

System description modules

System definitions are python **modules**. The modules include all state definitions, and the edges that connect the states:

```
from guardian import GuardState

prefix = 'SUS-MC2'

class SAFE(GuardState):
    ...

class DAMPED(GuardState):
    ...

edges = [...]
```

State definition

States are **class** definitions that inherit from the `GuardState` base class. Each `GuardState` includes the two state **methods** that are overriden by the user to program state behavior:

```
class DAMPED(GuardState):

    # This function is executed once
    def main(self):
        ...

    # This function is executed in a loop
    def run(self):
        ...
```

State execution model

The execution model of a state is straightforward, e.g.:

```python
# initialize state object
state = system.DAMPED()

# execute main state code
state.main()

# execute run state code in loop
# until it returns True
while True:
    status = state.run()
    if status is True:
        break
```

Edges

Directed edges between states are specified in the `edges` variable as a *list* of *tuples* of the form (FROM_STATE, TO_STATE):

```
edges = [
    ('DAMPED', 'ALIGNED'),
    ]
```

goto states, i.e. states with implicit edges coming from every other state in the graph, are specified by adding the 'goto' flag in the state definition, e.g.:

```
class SAFE(GuardState):
    goto = True
```

Jump transitions

If a state method returns a string it is interpreted as a state name and Guardian immediately transitions to that state. This is known as a **jump transition**:

```
class ALIGNED(GuardState):
    def run(self):
        if is_watchdog_tripped():
            return 'TRIPPED'
```

Support code and importing

Modules can include arbitrary other function/class/variable definitions.

```
def helper_function():
    ...

class DAMPED(GuardState):
    def main(self):
        helper_function()
```

Modules can also **import** other modules, or objects from other modules. For instance the SUS-MC2.py system description imports states from a base SUS.py suspension module:

```
from SUS import *
```

Ezca EPICS channel access

All channel access is done through the LIGO custom `Ezca` EPICS channel access interface.

The `ezca` object is available from anywhere in the system description module:

```
prefix = 'SUS-MC2'

class ALIGNED(GuardState):
    def main(self):
        ezca['M2_LOCK_L_GAIN'] = 10
```

Ezca EPICS channel access

If a `prefix` is specified in the system description module it will be combined with the local IFO variable to produce a proper channel prefix that is then passed to the `Ezca` object upon initialization. E.g.:

```
prefix = 'SUS-MC2'
```

becomes:

```
ezca = Ezca( 'L1:SUS-MC2_' )
```

Further `ezca` calls then only need to reference the rest of the channel name:

```
ezca['M2_LOCK_L_GAIN']
```

Ezca EPICS channel access

Ezca includes the usual **read**/**write** methods (accessible via two forms):

```
ezca.read('M2_LOCK_L_GAIN')
ezca['M2_LOCK_L_GAIN']

ezca.write('M2_LOCK_L_GAIN', 10)
ezca['M2_LOCK_L_GAIN'] = 10
```

as well as the **switch** method for dealing with LIGO standard filter modules (SFM):

```
ezca.switch('M2_LOCK_L','FM1','ON')
```

(SFM interface is being improved to add more useful methods.)

Timers

Timers can be used to measure off a specific amounts of time in a state. They are decremented every execution cycle. They should be used instead of issuing blocking `time.sleep()` calls when specific timeout conditions can't be tested for explicitly.

Set up the timer in the `main()` function by giving it a name and specifying the amount of time in seconds:

```
    def main(self):
        self.timer['mytimer'] = 2
```

The timer will automatically count down and will return True after it reaches zero:

```
    def run(self):
        if self.timer['mytimer']:
            do_something()
```

Other available methods

State can write to the guardian log:

```
    log('something is happening')
```

There are also special methods for manager nodes to interact with their subordinates:

```
    self.node['IMC'] = 'LOCKED'
    if self.node['IMC'] == 'LOCKED':
        ...
```

WARNING: this interface will likely be changed/improved in the near future.

example: IMC device guardian module code

```python
# -*- mode: python; tab-width: 4; indent-tabs-mode: nil -*-

from guardian import GuardState

##################################################

prefix = 'IMC'

##################################################

lockthreshold = 800

##################################################

# initial request state
request = 'LOCKED'

class INIT(GuardState):
    def run(self):
        if self.ezca['MC2_TRANS_SUM_INMON'] < lockthreshold:
            return 'ACQUIRE'
        else:
            return 'LOCKED'

class ACQUIRE(GuardState):
    goto = True

    def main(self):
        self.ezca['REFL_SERVO_IN1GAIN'] = -10
        self.ezca['REFL_SERVO_COMBOOST'] = 0

    def run(self):
        if self.ezca['MC2_TRANS_SUM_INMON'] < lockthreshold:
```

The Guardian interface

Guardian programs

Guardian includes four programs:

guardian
: Core guardian daemon program. Executes system state machines, of single states and paths.

guardctrl
: Interface to the site Guardian infrastructure, for controlling nodes and accessing logs.

guardmedm
: Launch the MEDM control interface for a Guardian node.

guardutil
: Utility program for displaying information about systems.

Guardian system identifiers

All programs accept system description module names, e.g. 'SUS_MC2', as their primary argument.

They look for the modules in Guardian-specific USERAPPS paths:

```
$USERAPPS/<subsystem>/<site>/guardian
$USERAPPS/<subsystem>/common/guardian
```

Guardian daemon

The core guardian program is the `guardian` daemon. It loads the system module and executes the state machine described therein. It has three modes of operation:

```
guardian [<options>] <module>
guardian [<options>] <module> <state>
guardian [<options>] <module> <state> <request>
guardian [<options>] [-i <module>]
```

Guardian daemon

```
guardian [<options>] <module>
```

When given the name of a system description module, it loads the module and starts executing the state machine described therein.

It logs to stdout, and is controlled by the guardian medm interface (see `guardmedm` below).

Usually this mode would only be run through the main site infrastructure (see `guardctrl` below).

Guardian daemon: states and paths

```
guardian [<options>] <module> <state>
```

If `guardian` is called with a single additional state argument, the single state will be executed on it's own until it completes, at which point guardian exits.

```
guardian [<options>] <module> <state> <request>
```

If two states arguments are specified, it is interpreted as a *path* in the state graph. Guardian will attempt to execute the path, and will exit when it completes the requested state.

These can be executed directly from the command line, and are very useful protyping and debugging.

Guardian daemon: interactive shell

```
guardian [<options>] [-i <module>]
```

If no argument is given, or the "interactive" flag is specified, an interactive shell will be launched (guardian-special ipython):

```
controls 0$ guardian
____________________
aLIGO Guardian Shell
____________________
prefix: L1:

In [1]:
```

This is useful for testing commands, and interacting with ezca:

```
In [1]: ezca['SUS-MC2_M2_LOCK_L_GAIN']
Out[1]: 3
```

guardctrl

`guardctrl` is the main interface to the site infrastructure. It is used for controlling nodes running on the site guardian machines ({h1,l1}guardian0)

From this interface, new nodes can be created, started, stopped, restarted, etc.:

```
controls 0$ guardctrl create SUS_MC2
controls 0$ guardctrl start SUS_MC2
controls 0$ guardctrl stop SUS_MC2
```

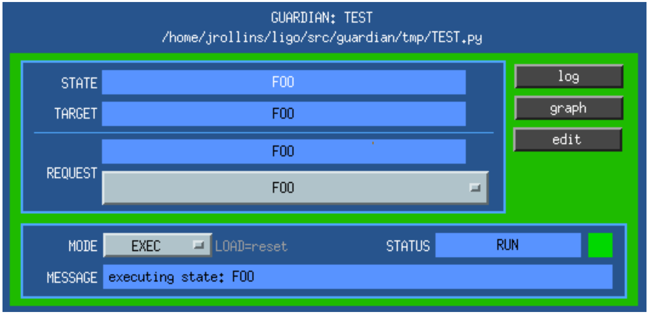
It can also be used to view node logs:

```
controls 0$ guardctrl log SUS_MC2
```

guardmedm

`guardmedm` launches the medm control interface to a specific guardian daemon:

```
controls 0$ guardmedm SUS_MC2
```



```
GUARDIAN: TEST
/home/jrollins/ligo/src/guardian/tmp/TEST.py
STATE    FOO
TARGET   FOO
REQUEST  FOO
         FOO
MODE     EXEC  LOAD=reset     STATUS  RUN
MESSAGE  executing state: FOO
```

It is used for controlling the daemon, requesting a state, accessing logs, displaying system graph, etc.

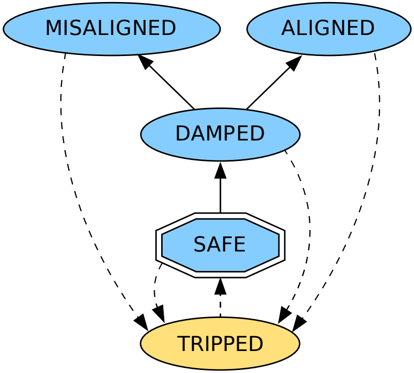
guardutil

`guardutil` has useful functions for developing systems.

```
controls 0$ guardutil SUS_MC2 graph
```

Among other things, it draws system graphs, which are very useful for understanding and debugging systems.

The graph drawings are still underdevelopment, and will continue to be improved to provide more useful info about the states.


MISALIGNED
ALIGNED
DAMPED
SAFE
TRIPPED