Creating a State-Synchronized Multi-Processes Software Architecture in ROS and Python

David Kooi

[1]Department of Science and Engineering, University of California Santa Cruz, 1156 High St, Santa Cruz, CA 95064

*To whom correspondence should be addressed; E-mail: dkooi@ucsc.edu.

In Object Oriented Programming, sharing state between objects is commonly solved using the Observer design pattern.(1) This paper is a project report on how the Observer pattern was used to enable a state-synchronized multi-process software architecture within the Robotic Operating System(ROS)(2) ecosystem. This paper covers the design rational, implementation, and ROS integration of this multi-process capability. This architecture serves as the basis for the Topcon Corporation sponsored University of California Santa Cruz Computer Engineering undergraduate senior design project for autonomous bulldozer research and development. The project name is Sambuca.

This article is a Project Report and is designed to report the project software development to the Topcon Corporation and to our senior design advisor. It is also intended to act as documentation. Results and application may also be of interest to general academic, scientific, and corporate audiences interested in software engineering.

Contents

List of Figures

1 Introduction

Our undergraduate senior design project involves automating a bulldozer's digging operation. To structure our application development we used the Robotic Operating System(ROS)(2) developed by Open Robotics. ROS provides a large library of robotic middleware ranging from camera drivers to simulation tools.

We are developing automation software on a scaled bulldozer model dubed the 'Housecat'.

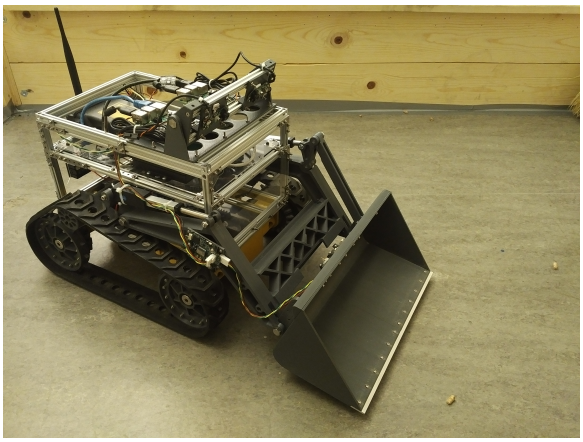

Figure 1: Topcon Housecat

The Housecat receives information about the surroundings through one monocular camera and one set of stereo cameras. They are used for two-dimensional sensing and three dimensional sensing, respectively. One problem of automation is how to aggregate multiple sources of information to achieve a decision.

Commonly a decision structure is provided by a state machine. ROS supports a state machine called Smach(3). A user is allowed to define states, events, and transitions. However, the Smach structure was too limited for our organizational desires.

As a result, we developed an Observer(1) based software architecture to manage state events

and transitions between multiple Control-Processes.

2 Control-Processes

It is important to define Control-Process. In the context of our automation project, Control-Process refers to the application-level software module that is making movement decisions and updating the system with events. In this context a control-process *does not* mean:

- A feedback controller
- A Unix process with a 5 digit PID
- A software driver

In this context a control-process *does* mean:

- A python software object responsible for making robotic decisions and publishing events

For example, in the Sambuca project, we have two primary data streams:

1. Monocular Camera Frames(2D)
2. Stereo Camera Frames(3D)

We define a **Process_2D** and a **Process_3D** to handle the respective frames and to make movement decisions. The next section explains the desired Control-Process characteristics.

3 Structural Desires

We have identified several structural desires for our Control-Processes:

- Modularity
- Strict Seperation
- Time Independence
- State Machine Compatibility
- ROS Compatiblity

3.1 Modularity

A Control-Process must be modular. For example, the Sambuca project uses a stereo-camera system to generate a pointcloud. A LIDAR system also generates a point cloud. A modular **Process_3D** would enable quick swapping of stereo-camera and LIDAR hardware with minimal changes to the software.

3.2 Strict Separation

A Control-Process must be strictly separated. This means that if the changes need to be made to the handling and decision making of the 2D data, the section of code responsible for the 3D data handling does not need to be touched.

3.3 Time Independence

A Control-Process must run independent of whether another control-process has completed it's task. That is, a control-process should not have to wait on another control-process.

3.4 State Machine Support

A Control-Process should be run as a state machine. This means states, events, and transitions should be defined for a control-process.

3.5 ROS Compatible

A Control-Process must work with ROS. ROS is the middleware Sambuca is built on and all software developed must be compatible with ROS.

4 A Singular State Machine

Traditionally a state machine is singular. All behavior, for example, 3D behavior and 2D behavior, occur within a state.

Figure 2 shows a conflict between strict separation and state machine support. If we change how Process2D behaves in StateA, we need to touch code near the behavior of Process3D. In actual implementation, object use and behavior become more complex. Efforts can be made to encapsulate object state behavior, but there is still the problem of time independence. Process3D must wait until Process2D is finished. This causes problems when performing intensive computation like neural-net feature recognition or point-cloud analysis. If these computations are performed serially, the reaction time depends on the sum of all computations. We wish to

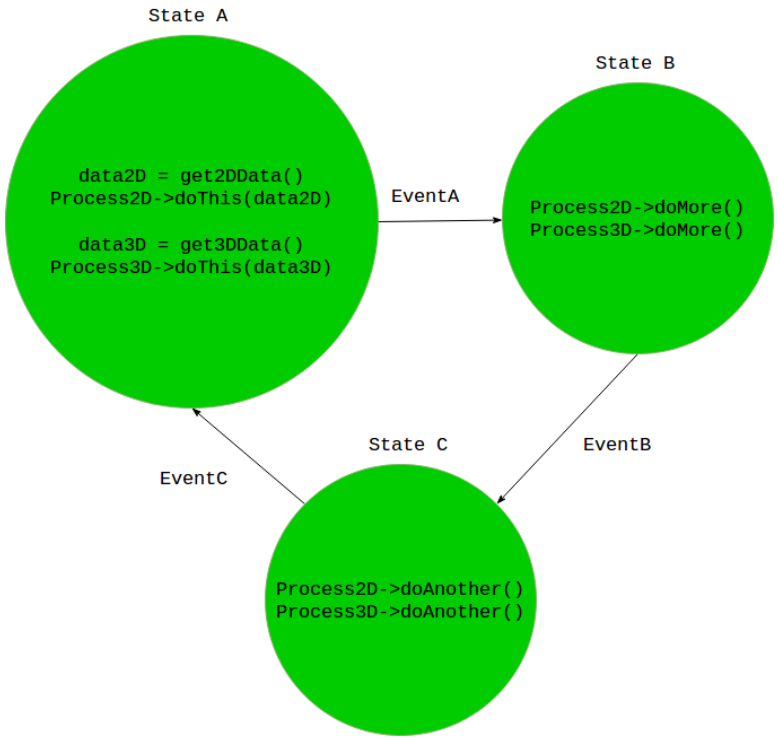

Figure 2: Non-Separation of Concerns

perform computations in parallel because then the reaction time depends on the longest computation.

A couple questions arise:

- How can a separation of concerns be accomplished if the concerns are grouped together in a state?
- What can be done to allow each control-process to run without waiting?

To achieve these architectural desires we look to proven OOP design patterns.

5 Design Rational

The challenge is to combine these desires into a well structured software architecture. Luckily, Erich Gamma et al, has compiled a selection of canonical object oriented design patterns in his book: Design Patterns: Elements of Reusable Object-oriented Software.(1) One particular pattern fits Sambuca's case.

Gamma says that the Observer pattern:

> "Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically."(1)

Taking from the Observer pattern we can fulfill our structural desires.

5.1 Fulfillment of Structural Desires

1. (3.1) If we define an abstract control-process class, we can enforce a a common object interface. If the inputs and outputs are the same, the internal object behavior can be arbitrary, and we have achieved **modularity**.

2. (3.2) **Strict separation** is also accomplished because each control-process is only concerned with it's own state implementation.

3. (3.3) If each control-process keeps internal state, kept updated by the Observer to the reference state machine, each control-process can operate simultaneously while in the same state. This achieves **time independence**.

4. (3.4) The idea of state is explicit in the description of the Observer pattern: "If one object changes state, all dependent objects change state...." If we define a one-to-many dependency between one 'reference state machine' and many control-processes, each

control-process can maintain it's own states while keeping in sync with the reference. This achieves **state machine** support.

5. (3.5) **ROS Compatibility** is left as an exercise in implementation. See Section (6)

5.2 Top-Level Design

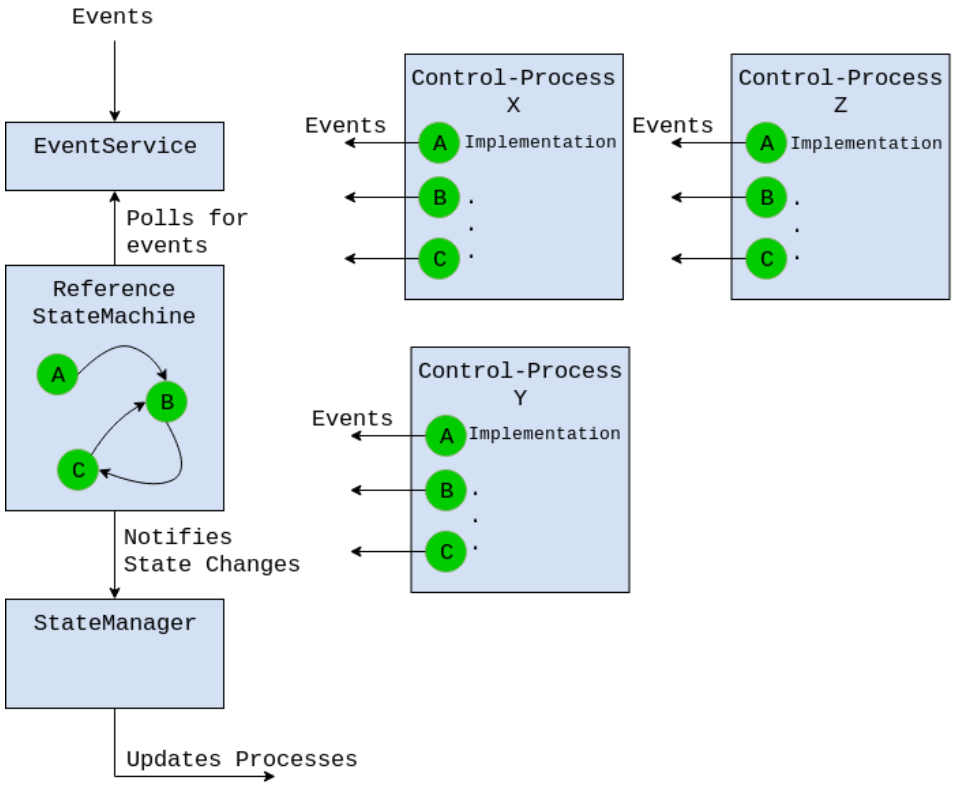

Figure 3: Top-Level Block Diagram

Figure 3 shows a design that achieves the goals 3.1, 3.2, 3.3, 3.4 and 3.5.

A reference state machines consumes events and performs the appropriate state transitions. Each control-process observers the StateManager. The StateManager updates all control-processes when to change states.

All control-processes run in parallel because each control-process implements it's own states. Each control-process also emits events that can cause transitions in the reference state machine.

6 Observer UML Design

The first step is to create a class diagram of the observer design. This will guide the implementation.

6.1 UML Design Template

The canonical observer pattern shown in Figure 4 will be used to create our own implementation.

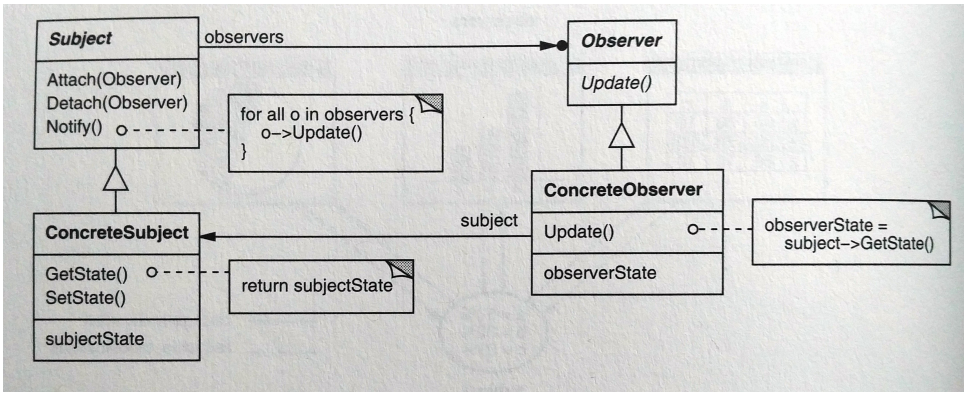

Figure 4: UML Relations of the Canonical Observer Pattern (1)

7 Sambuca UML Design and Implementation

Figure 5 shows an abstracted view of the design. Sections 7.1 and 7.2 cover the design and implementation of the control-process. Sections 7.3 and 7.4 cover the design and implementation of the reference state machine. Section 7.5 covers the implementation of the StateManager. section 7.6 details the implementation of the EventService. Finally, Section 8 gives a simple example of how these components are connected and run.

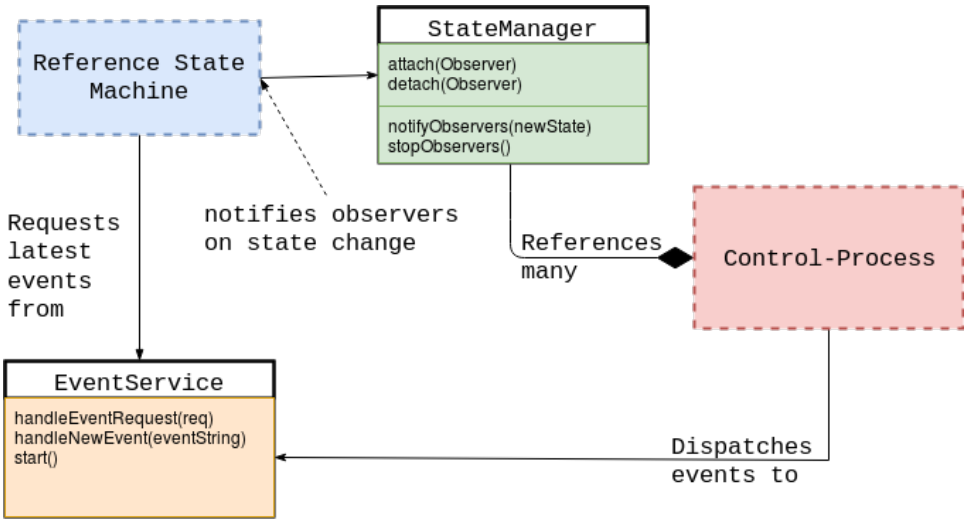

Figure 5: UML Relations for Sambuca Observer Design

7.1 Control-Process Design

Note that in figure 5 the Control-Process is-a StateObserver. That is, it observes the state of the reference state machine. Observation is facilitated by the StateManager. The Control-Process references many ObserverStates. That is, each Control-Process maintains it's own state implementations. The ObserverState is an abstract state and each Control-Process needs to implement one ObserverState for each ReferenceState(7.3 defined. A **runState()** method is implemented by the user and contains Control-Process specific actions for a given state. These state implementations are python threads. The extension of python threading allows multiple Control-Process States to run 'concurrently' as shown in Figure 5. In this paper, Observer-States will be also be referred to as **state-threads** and **state-implementations**.

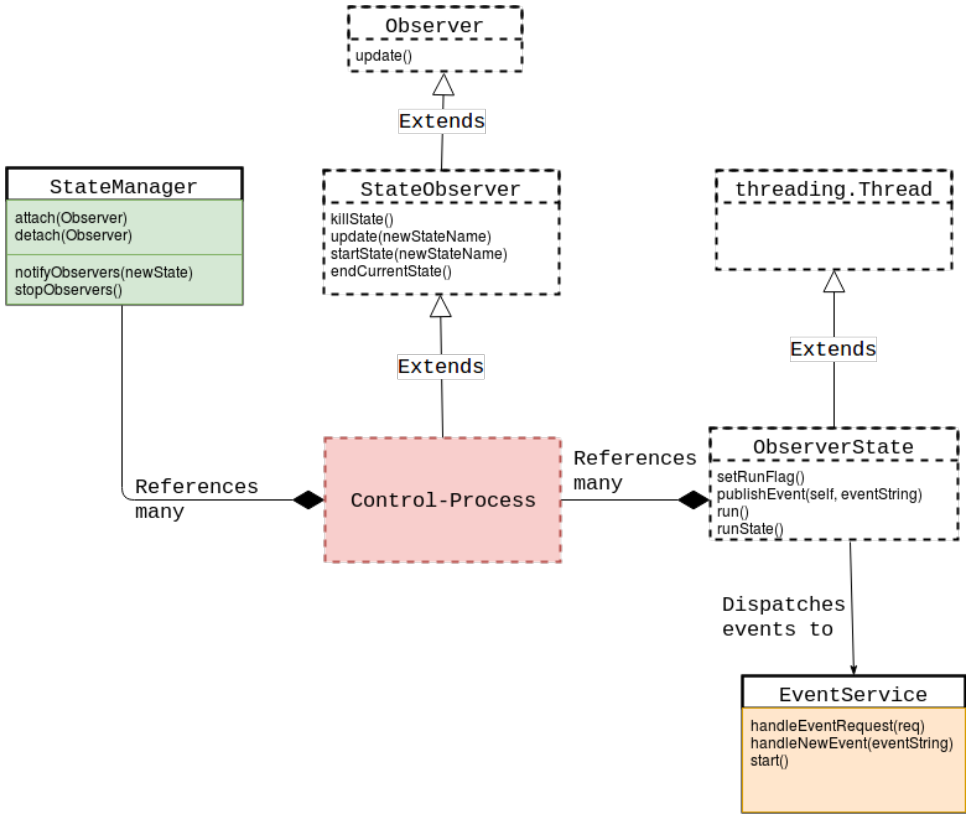

Figure 6: UML Relations for Sambuca Control Process

7.1.1 Python Threading Limitations

The reader should be aware that Python threading is not truly concurrent. Rather, python threads are sequentially executed. Threads must obtain a Global Interpreter Lock(GIL) before running. According to the Python Software Foundation:

> "...[a] rule exists that only the thread that has acquired the GIL may operate on Python objects or call Python/C API functions."(4)

A way around the Python GIL is to use the Python multiprocessing library. This allows an application to fully use multiple processors on a machine. To change this architecture to support multiprocessing, one would change the ObserverState parent from threading.Thread to multiprocessing.Process.

7.2 Control-Process Implementation

7.2.1 StateObserver Implementation

The StateObserver is initalized with a name and a list of ObserverStates. The **runFlag** on line 85 provides communication between the StateObserver and it's ObserverStates. Recall that an ObserverState is a python thread. The way Python threads operate add consideration when staring and changing states. A Python thread object cannot be restarted. The StateObserver must keep references to all ObserverState classes. The required state-thread setup in started on line 93. A reference to the ObserverState class and an instantiated ObserverState object are both saved in the **myStates** dictionary.

```
75 class StateObserver(object):
76
77     def __init__(self, name, myStates):
78         """
79         @param myStates: A list of instantiated and uninstantiated
80         ObserverState dictionaries. Use the factory function CreateObserverState
81         to create the required object dictionary.
82         """
83         self.name           = name
84         self.myStates       = dict()
85         self.runFlag        = threading.Event()
86         self.currentState   = None
87         self.isRunning      = False
88
89         # Setup ObserverState objects
90
91         # 1. Add object and instantiated state to our dictionary
92         # 2. Set run flag for all ObserverState objects
93         for state in myStates:
94             stateName = state[ObsSettings.STATE].name
95             # 1.
96             self.myStates[stateName] = dict()
97             self.myStates[stateName][ObsSettings.OBJECT] = state[ObsSettings.OBJECT]
98             self.myStates[stateName][ObsSettings.STATE]  = state[ObsSettings.STATE]
99             # 2.
100            self.myStates[stateName][ObsSettings.STATE].setRunFlag(self.runFlag)
```

Figure 7: StateObserver Implementation(1)

The **startState()** method allows a new state to be loaded. The **runFlag** is used by all the state-threads to signal whether to continue running or to stop. The StateObserver resets the **runFlag** which gives the new state permission to run.

```
126     def startState(self, newStateName):
127         '''
128         - Replaces the current state with a new and instantiated state object.
129         - Starts the current state
130 
131         @param newStateName: The new state to run.
132         '''
133 
134         # Load new state
135         self.currentState =\
136         self.myStates[newStateName][ObsSettings.STATE]
137 
138         # Reset run flag
139         self.runFlag.set()
140 
141         # Start new state
142         self.currentState.start()
143         self.isRunning = True
```

Figure 8: StateObserver Implementation(2)

Figure 9 shows the **endCurrentState()** method. The **runFlag** is cleared on line 157 and the state-thread is joined. Lines 166 - 177 involve thread object maintenance.

```
146     def endCurrentState(self):
147         '''
148         - Clears the run flag and waits for the state to finish
149         - Since thread objects can only be used once, instantiate a new
150           state thread object and assign it back to the myStates dictionary
151         '''
152         # Nothing to do if first run
153         if self.currentState is None:
154             return
155 
156         # Unset run flag and wait for state to finish
157         self.runFlag.clear()
158         self.currentState.join(STATETIMEOUT)
159 
160         # Throw error is we timed out
161         if(self.currentState.isAlive()):
162             stateName = self.currentState.name
163             raise RuntimeError("State thread {}\
164                     timed-out in {}".format(stateName, self.name))
165 
166         # Reset state thread by:
167         # 1. Creating a new thread object
168         name       = self.currentState.name
169         parentName = self.name # This StateObserver is parent to the new thread
170         self.myStates[name][ObsSettings.STATE] =\
171         self.myStates[name][ObsSettings.OBJECT](parentName, name)
172         # 2. Setting the run flag
173         self.myStates[name][ObsSettings.STATE].setRunFlag(self.runFlag)
174 
175         # Set isRunning flag to let state manager know we are finished with
176         # ending the state
177         self.isRunning = False
```

Figure 9: StateObserver Implementation(3)

7.2.2 OberverState Implementation

From Figure 6 we see that the ObserverState extends threading.Thread. We also see that ObserverState is abstract. The **runState()** method on line 68 must be implemented. On line 60 we see that the **runState()** method will repeatedly run while the **runFlag** is set.

The **evPub** and **actionClient** references allow the ObserverState to communicate with the larger ROS framework. Also note the events being published from each Control-Process in Figure 3. Those events are published from each state-thread using the **publishEvent()** method on line 55.

```
30 class ObserverState(threading.Thread):
31     """
32     Abstract ObserverState.
33 
34     Each StateObservermust maintain one ObserverState for every
35     ReferenceState held by the StateManager.
36     """
37     def __init__(self, parent, name):
38 
39         super(ObserverState, self).__init__(name=name)
40 
41         self.evPub = rospy.Publisher(SambucaTopics.NewEventTopic, String,\
42                                      queue_size=10)
43 
44         self.actionClient =\
45         actionlib.SimpleActionClient(SambucaActionServerName,\
46                                      movementAction)
47         #rospy.init_node(name, anonymous=True)
48 
49         self.parent  = parent
50         self.runFlag = None
51 
52     def setRunFlag(self, runFlag):
53         self.runFlag = runFlag
54 
55     def publishEvent(self, eventString):
56         self.evPub.publish(eventString)
57         time.sleep(0.5)
58 
59     def run(self):
60         if(self.runFlag is None):
61             raise AttributeError\
62             ("StateObserver must set the runFlag for all ObserverStates")
63 
64         while(self.runFlag.isSet()):
65             self.runState()
66 
67 
68     def runState(self):
69         raise NotImplementedError
```

Figure 10: ObserverState Implementation(1)

7.3 Reference State Machine Design

The Reference State Machine is an implementation of the smach(3) state machine library. A state machine is built using states, transitions, and events. The ReferenceState class extends **smach.State** and adds several capabilities important to working with the StateManager and EventService.

The **smach.StateMachine** works by referencing many **smach.States**. When the current **smach.State** receives an event it is forwarded to the **smach.StateMachine** . The **smach.StateMachine** performs the transition and executes the next **smach.State**.

The each ReferenceState maintains a reference to the StateManager. This allows the ReferenceState to call **notifyObservers()** when a new event is received.

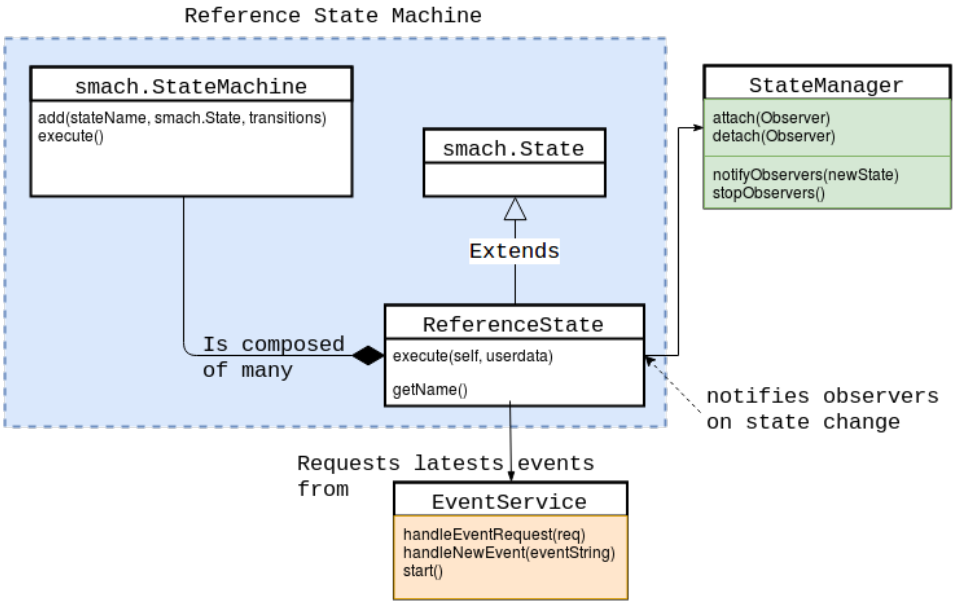

Figure 11: UML Relations for Sambuca Reference State Machine

7.4 Reference State Machine Implementation

The ReferenceState is created with a name, transitions, outcomes, and a reference to the StateManager. A ROS proxy to the EventService is created in order to poll for new events. The **execute()** method is called by the **smach.StateMachine**. A ReferenceState will continuously poll for a new event during execution. If a new event is received the new state is sent the StateManager is used to notify the StateObservers. The event is returned to let the **smach.StateMachine** transition.

```
61 class ReferenceState(smach.State):
62     def __init__(self, name, transitions, outcomes, stateManager):
63         smach.State.__init__(self, outcomes = outcomes)
64 
65 
66         self.name = name
67         self.transitions = transitions
68         self.stateManager = stateManager
69 
70         # Create proxy for NewEvent service
71         self.pollEvents = rospy.ServiceProxy(SambucaServices.NewEventService,\
72                 NewEvent, persistent=True)
73 
74 
75     def execute(self, userdata):
76         """
77         Run the reference state by constantly polling for new events.
78         """
79         rospy.logdebug("Entering {} reference state".format(self.name))
80 
81         # Is there a better way to spin?
82         while True:
83 
84             # Wait for a new event
85             try:
86                 rospy.wait_for_service(SambucaServices.NewEventService)
87 
88                 resp  = self.pollEvents()
89                 event = resp.event
90                 rospy.logdebug("{} Receieved event {}".format(self.name, event))
91             except rospy.ServiceException as e:
92 
93                 # Something bad happened
94                 print("EventService did not process request: " + str(e))
95 
96                 # Try again
97                 continue
98             except rospy.ROSInterruptException:
99                 # We are exiting, stop all observer state threads
100                self.stateManager.stopObservers()
101
102            # Do nothing if NOOP
103            if(event == StateEvents.NOOP):
104                rospy.logdebug("{} received NOOP".format(self.name))
105                continue
106            else:
107                # Get the new state and update the observers
108                newState = self.transitions[event]
109                self.stateManager.notifyObservers(newState)
110
111                # Returning the event changed the Smach state
112                return event
```

Figure 12: ReferenceState Implementation

7.5 StateManager Implemenation

An important part of the Observer pattern is the notification of Observers. Figure 5 shows that the Reference State Machine has access to the StateManager. The Reference State Machine may call **notifyObservers()** when it receives a state-changing event. Many state-implementations may be running at the same time. The first action is to end all current states. (Line 24).

The StateManager should not notify observers of a new state until all state threads are finished. Or example, some Control-Process may be performing an image analysis that take hundreds of milli-seconds to finish. To keep Control-Process synchronization, the StateManager waits for state threads to finish.

This is done by polling the **isRunning** flag of all observers and checking that all flags are False. (Line 33)

Once all state implementations have finished the observers are given a new state. (Line 43).

Note how the central management of obsrevers on line 24 and line 43 follow the design pattern Subject class in Figure 4.

```
11 class StateManager(object):
12     def __init__(self):
13         self.observers = []
14 
15         rospy.init_node("StateManager", \
16                 log_level=sambuca_settings.log_level)
17 
18 
19     def notifyObservers(self, newState):
20         """
21         Notify all observers of a state change
22         """
23         # End the currently running state
24         for observer in self.observers:
25             rospy.logdebug("SM Ending observer states")
26             observer.endCurrentState()
27 
28         # Spin while we wait for all states threads to have joined
29         rospy.logdebug("SM Waiting for states to finish...")
30         statesRunning = True
31         while(statesRunning):
32             # Get all states into a list
33             statusList = [observer.isRunning for observer in self.observers]
34 
35             # Check if all states are False
36             statesRunning = not all(status == False for status in statusList)
37 
38 
39         rospy.logdebug("SM States finished, starting new state\
40                 {}".format(newState))
41 
42         # All observers are ready to accept a new state
43         for observer in  self.observers:
44             observer.startState(newState)
```

Figure 13: State Manager Implementation(1)

A **stopObservers()** method is defined on line 46. This is used during shutdown to notify all state-threads to quit.

Lastly an **attach()** and **detach()** method are defined in order to allow connection/disconnection of an observer.

```
46      def stopObservers(self):
47          """
48          Cause all Observers to stop their running states
49          """
50          rospy.loginfo("SM Stopping observer states")
51          for observer in self.observers:
52              observer.killState()
53
54      def attach(self, observer):
55          self.observers.append(observer)
56
57      def detach(self, observer):
58          self.observers.remove(observer)
```

Figure 14: State Manager Implementation(2)

7.6 Event Service Implementation

The EventServer consumes events from the Control-Process state-threads. It also serves the events to the ReferenceStates. **HandleEventRequest()** handles the service request and **HandleNewEvent** is invoked when a new event is received.

```
13 class EventServer(object):
14     def __init__(self):
15         rospy.init_node("SambucaEventServer",\
16                 log_level=sambuca_settings.log_level)
17 
18         # By default we are waiting for events
19         self.newEvent = None
20 
21         # Create a service to deliver new events to the StateManager
22         rospy.Service(SambucaServices.NewEventService, NewEvent,\
23                 self.HandleEventRequest)
24 
25         # Create subscriber to consume events generated by StateObservers
26         rospy.Subscriber(SambucaTopics.NewEventTopic, String,\
27                 self.HandleNewEvent)
28 
29 
30     def HandleEventRequest(self, req):
31         """
32         Block until a newEvent string is received by
33         the subscriber.
34         """
35 
36         while(self.newEvent is None):
37             pass
38 
39         # Record new event
40         eventString = self.newEvent
41         # Reset newEvent
42         self.newEvent = None
43         return NewEventResponse(eventString)
44 
45     def HandleNewEvent(self, eventString):
46         rospy.logdebug("EventServer received event:\
47                 {}".format(eventString.data))
48         self.newEvent = eventString.data
49 
50     def start(self):
51         rospy.spin()
52 
53 if __name__ == "__main__":
54     e = EventServer()
55     e.start()
```

Figure 15: Event Service Implementation

8 Example Application and Discussion

A simple example is presented in this section. The example shows all the components implemented, created and connected. Figure 16 shows the example application's states, events, and transitions. Figure 17 shows two Control-Processes within this application. Each Control-Process uses three state-implementations.

This section includes two examples of how this software architecture enables modularity. This is intended as a discussion piece.

8.1 Example State Sequence

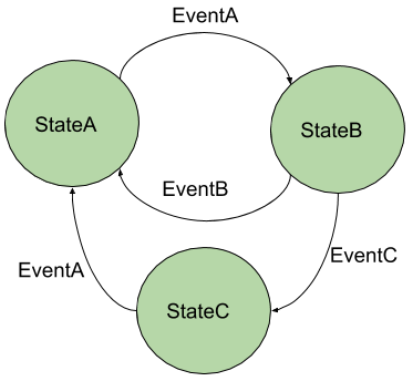

Figure 16: Example State Machine

8.2 Process Diagram

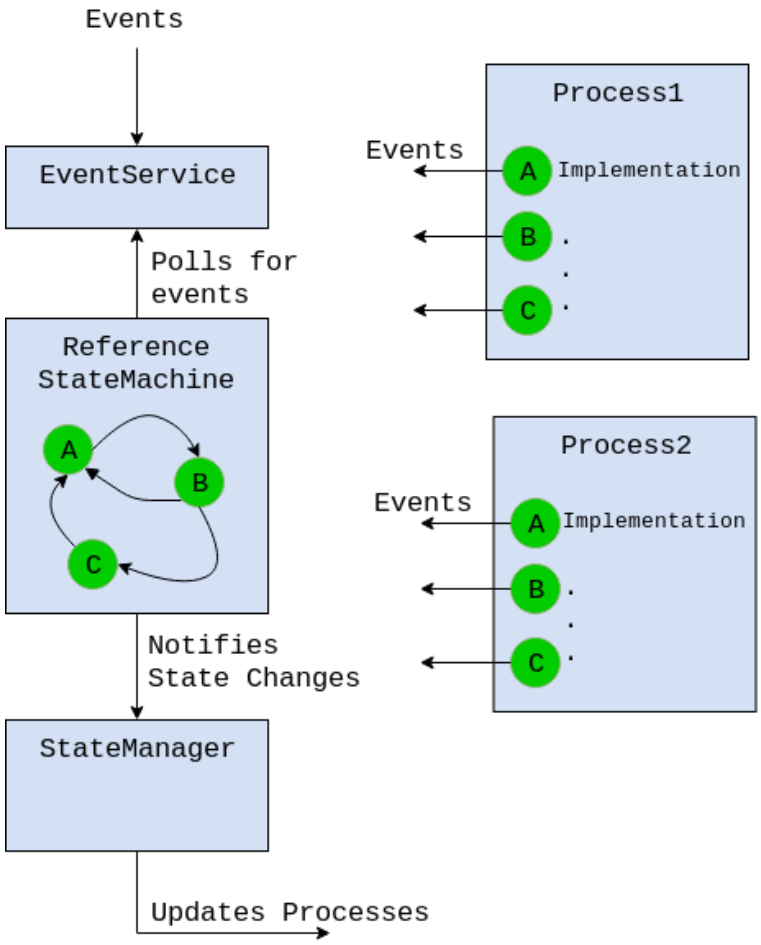

Figure 17: Example Process Diagram

8.3 State Definitions

Events, transitions, and states are defined within a python file.

```
4  StateEvents = enum(NOOP="NOOP", EVENT_A="EventA", EVENT_B="EventB",\
5                     EVENT_C="EventC")
6
7  StateNames = enum(STATE_A="StateA", STATE_B="StateB", STATE_C="StateC")
8
9
10 # Keep transition dictionaries 'private'
11 # Provide access through the transition enum
12 _StateA_Transitions = {StateEvents.EVENT_A:StateNames.STATE_B}
13 _StateB_Transitions = {StateEvents.EVENT_B:StateNames.STATE_A,\
14                        StateEvents.EVENT_C:StateNames.STATE_C}
15 _StateC_Transitions = {StateEvents.EVENT_A:StateNames.STATE_A}
16
17 _StateA_Outcomes    = [StateEvents.EVENT_A]
18 _StateB_Outcomes    = [StateEvents.EVENT_B, StateEvents.EVENT_C]
19 _StateC_Outcomes    = [StateEvents.EVENT_A]
20
21
22
23 StateTransitions = enum(STATE_A=_StateA_Transitions,\
24                         STATE_B=_StateB_Transitions,\
25                         STATE_C=_StateC_Transitions)
26
27 StateOutcomes    = enum(STATE_A=_StateA_Outcomes,\
28                         STATE_B=_StateB_Outcomes,\
29                         STATE_C=_StateC_Outcomes)
```

Figure 18: State Machine Definition

8.4 State Implementation

The example application re-uses an **EchoState** to prove 'concurrency'. The **EchoState** prints it's parent Control-Process(self.parent), what state it represents(self.name), and a count.

```
5  class EchoState(ObserverState):
6      """
7      A test state that periodically prints it's name
8      and count
9      """
10
11     def __init__(self, parent, name):
12         super(EchoState, self).__init__(parent, name)
13         self.count = 0
14
15     def runState(self):
16         print("[{}] {} {}".format(self.parent, self.name, self.count))
17         self.count += 1
18         sleep(1)
```

Figure 19: Example State Machine

8.5 Startup Script: Process Creation

Figure 20 shows two Control-Processes being created. First a StateManager is created on line 20. Next Process1 and Process2 are created with the **EchoState** implementation. The **CreateObserverState()** function takes arguments: parentName, stateName and stateClass. parentName and stateName are both strings and stateClass is a reference to the state implementation definition. (In this case **EchoState**)

Each process much be attached to the StateManager as part of the Observer pattern.

```
17 def main():
18
19     # State Manager
20     stateManager = StateManager()
21
22     # Independent State-Observing Processes
23     # PROCESS 1
24     processName = "Process1"
25     obs_StateA = CreateObserverState(processName, StateNames.STATE_A, EchoState)
26     obs_StateB = CreateObserverState(processName, StateNames.STATE_B, EchoState)
27     obs_StateC = CreateObserverState(processName, StateNames.STATE_C, EchoState)
28
29     stateList = [obs_StateA, obs_StateB, obs_StateC]
30     Process_1 = StateObserver(processName, stateList)
31
32     # Attach Process to StateManager
33     stateManager.attach(Process_1)
34
35
36     # PROCESS 2
37     processName = "Process2"
38     obs_StateA = CreateObserverState(processName, StateNames.STATE_A, EchoState)
39     obs_StateB = CreateObserverState(processName, StateNames.STATE_B, EchoState)
40     obs_StateC = CreateObserverState(processName, StateNames.STATE_C, EchoState)
41
42     stateList = [obs_StateA, obs_StateB, obs_StateC]
43     Process_2 = StateObserver(processName, stateList)
44
45     # Attach Process to StateManager
46     stateManager.attach(Process_2)
```

Figure 20: Example Application Process Declaration

8.5.1 An example of modularity

This is an example of **modularity**. The state implementation used is arbitrary and may be swapped with another state implementation. In section 3.1 makes an example about hardware changes. Imagine if Process1 was responsible for handling three-dimensional point-clouds. Suppose Process1 was developed for a stereo camera. Now, it is decided that a LIDAR would

enable better decision making. The synchronized-state observer architecture enables the application developer to plug-in new LIDAR state implementations directly into the startup script.

8.6 Startup Script: Reference State Machine Creation

First a **smach.StateMachine** is created. Next three ReferencesStates are made. Note these all have a reference to the StateManager. We add the ReferenceStates to the **smach.StateMachine** on line 63.

On line 75 and 76 we update both processes with the same state. This kicks off the state-threads. Line 81 starts the ReferenceStateMachine and which blocks until the end of application: Usually a ROS interruption.

```
50      # Smach Reference States
51      sm = smach.StateMachine(outcomes=[])
52
53      ref_StateA = ReferenceState(StateNames.STATE_A, StateTransitions.STATE_A,\
54              StateOutcomes.STATE_A, stateManager)
55
56      ref_StateB = ReferenceState(StateNames.STATE_B, StateTransitions.STATE_B,\
57              StateOutcomes.STATE_B, stateManager)
58
59      ref_StateC = ReferenceState(StateNames.STATE_C, StateTransitions.STATE_C,\
60              StateOutcomes.STATE_C, stateManager)
61
62      # Add the ref_States to the smach StateMachine
63      with sm:
64          smach.StateMachine.add(StateNames.STATE_A, ref_StateA,\
65                  transitions=StateTransitions.STATE_A)
66
67          smach.StateMachine.add(StateNames.STATE_B, ref_StateB,\
68                  transitions=StateTransitions.STATE_B)
69
70          smach.StateMachine.add(StateNames.STATE_C, ref_StateC,\
71                  transitions=StateTransitions.STATE_C)
72
73
74      # Enter the first observer state
75      Process_1.update(StateNames.STATE_A)
76      Process_2.update(StateNames.STATE_A)
77
78      # Start state machine
79      # This spins until interrupt
80      try:
81          sm.execute()
82      except rospy.ROSInterruptException:
83          pass
84
```

Figure 21: Example Application Reference State Machine Declaration

8.6.1 An example of modularity

Another example of modularity is being able to change the initial state by updating the Control-Processes with a different initial state. (Note: The initial state of the Control-Processes must match the first state added to the **smach.StateMachine**.)

8.7 Event Generation

State machines are event driven so there needs to be something publishing events. The **EchoState** just prints it's name. The simple event generator shown in Figure 22 was made to drive the state machine.

In a real application events would be generated by the Control-Processes.

```
11 def ev_generator():
12     pub = rospy.Publisher(SambucaTopics.NewEventTopic, String, queue_size=10)
13     rospy.init_node("test_ev_gen")
14
15
16     events = [StateEvents.NOOP, StateEvents.EVENT_A, StateEvents.EVENT_B,\
17               StateEvents.NOOP, StateEvents.EVENT_A, StateEvents.EVENT_C,\
18               StateEvents.NOOP, StateEvents.EVENT_A]
19
20     rate = rospy.Rate(0.25) # 4 second period
21
22     while not rospy.is_shutdown():
23         for event in events:
24             pub.publish(event)
25             rate.sleep()
26
```

Figure 22: Example Application Event Generator

8.8 Example Application Output

Figure 23 shows the reference state machine receiveing EventA and both Control-Processes changing states to StateB. Both Control-Processes are printing their names and count's concurrently.

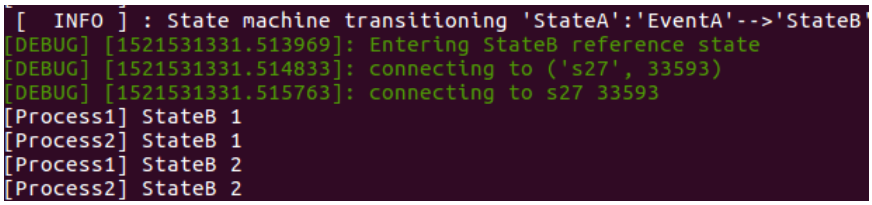

Figure 23: Example Application Output

9 Conclusion

The Object Oriented Observer Design Pattern has enabled the Sambuca software architecture to have:

- Modularity
- Strict Seperation
- Time Independence
- State Machine Compatibility
- ROS Compatibility

These structural desires have made it easier for our team to collaborate and have made development more flexible to change. We hope that the new software architecture will make professional development of the Sambuca application easier and more maintainable.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[2] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, 2009.

[3] J. Bohren. (2017) smach. http://wiki.ros.org/smach. Accessed: 2018-03-1.

[4] T. P. S. Foundation. (2017) Thread state and the global interpreter lock. https://docs.python.org/3/c-api/init.html#thread-state-and-the-global-interpreter-lock. Accessed: 2018-03-19.