machinist Documentation

Release 0.1.0

Jean-Paul Calderone

June 12, 2014

Contents

Contents:

CHAPTER 1

The State Machine Construction Toolkit

Machinist's aim is to make it easy to structure your code as an explicit state machine.

State machines are defined by supplying Machinist with four things:

1. A set of states.
2. A set of inputs.
3. A set of outputs.
4. A set of transitions.

State machines are represented by an object with a `receive` method which accepts an input object. The input object is either an object from the set of inputs or an object related to one of those objects in a certain way (more on that later). When an input is received by the state machine, its state is updated and outputs are generated according to the defined transitions.

If this sounds great to you then you might want to jump ahead to the *Basic Usage* documentation. Otherwise, read on.

CHAPTER 2

Benefits of Explicit State Machines

All software is an implementation of a state machine. The memory associated with a running program represents its current state. The executable code defines what transitions are possible. Myriad inputs and outputs exist in the form of:

- data read from file descriptors.
- signals or GUI events such as button clicks.
- data which is rendered onto displays.
- sound which is created by speakers.

The difference between an explicit state machine and software written without a state machine (let's call this *implicit state machine* software or *ism* software) mostly comes down to what it is easy to learn about the state machine being represented.

2.1 States

In the explicit state machine all of the states have been enumerated and can be learned at a glance. In ISM (implicit state machine) software it is impractical to enumerate the states: imagine a program with just one piece of memory, a 16 bit integer. There are 2^{16} (65536) states in this program. Without reading all the program that manipulates this state it's impossible to know which of them are important or how they might interact. Extend your imagination to any real piece of software which might operate on dozens, hundreds, or thousands of megabytes of memory. Consider the number of states this amount of memory implies. It's not just difficult to make sense of this collection of states, it is practically impossible.

Contrast this with an explicit state machine where each state is given a name and put in a list. The explicit state machine version of that program with a 16 bit integer will make it obvious that only three of the values (states) it can take on are used.

2.2 Inputs and Outputs

In the explicit state machine all of the inputs and outputs are also completely enumerated. In ISM software these are usually only defined by the implementation accepting or producing them. This means there is just one way to determine what inputs are accepted and what outputs are produced: read the implementation. If you're lucky, someone will have done this already and produced some API documentation. If you're doubly lucky, the implementation won't have changed since they did this.

Contrast this with an explicit state machine where the implementation is derived from the explicit list of inputs and outputs. The implementation cannot diverge because it is a function of the declaration.

2.3 Transitions

Once again, transitions are completely enumerated in the definition of an explicit state machine. A single transition specifies that when a specific input is received while the state machine is in a specific state a specific output is produced and the state machine changes to a specific new state. A collection of transitions completely specifies how the state machine reacts to inputs and how its future behavior is changed by those inputs. In ISM software it is conventional to define a constellation of flags to track the state of the program. It is left up to the programmer to build and remember the cartesian product of these flags in their head. There are also usually illegal flag combinations which the program is never *supposed* to encounter. These are either left as traps to future programmers or the implementer must take the tedious steps of building guards against them arising. All of this results in greater complexity to handle scenarios which are never even supposed to be encountered.

Contrast this with an explicit state machine where those flags are replaced by the state of the state machine. The valid states are completely enumerated and there is no need to look at a handful of flags to determine how the program will behave. Instead of adding complexity to handle impossible cases, those cases are excluded simply by *not* defining an explicit state to represent them.

CHAPTER 3

Basic Usage

State machines are constructed using `machinist.constructFiniteStateMachine`.

3.1 Inputs, Outputs, States

Before a machine can be constructed its inputs, outputs, and states must be defined. These are all defined using `twisted.python.constants`.

```
from twisted.python.constants import Names, NamedConstant

class Input(Names):
    FARE_PAID = NamedConstant()
    ARM_UNLOCKED = NamedConstant()
    ARM_TURNED = NamedConstant()
    ARM_LOCKED = NamedConstant()

class Output(Names):
    ENGAGE_LOCK = NamedConstant()
    DISENGAGE_LOCK = NamedConstant()

class State(Names):
    LOCKED = NamedConstant()
    UNLOCKED = NamedConstant()
    ACTIVE = NamedConstant()
```

3.2 Transitions

Also required is a transition table. The transition table is defined using `machinist.TransitionTable`. `TransitionTable` instances are immutable and have several methods for creating new tables including more transitions.

```
from machinist import TransitionTable

table = TransitionTable()
```

First, define how the `FARE_PAID` input is handled in the `LOCKED` state: output `DISENGAGE_LOCK` and change the state to the `ACTIVE`.

```
table = table.addTransition(
    State.LOCKED, Input.FARE_PAID, [Output.DISENGAGE_LOCK], State.ACTIVE)
```

Next, define how the `ARM_TURNED` input is handled in the `UNLOCKED` state: output `ENGAGE_LOCK` and change the state to `ACTIVE`.

```
table = table.addTransition(
    State.UNLOCKED, Input.ARM_TURNED, [Output.ENGAGE_LOCK], State.ACTIVE)
```

Last, define two transitions at once for getting out of the `ACTIVE` state (in this model `DISENGAGE_LOCK` and `ENGAGE_LOCK` activate a physical device to change the lock state; the state machine then waits for an input indicating the physical device has completed the desired operation).

`addTransitions` is a convenient way to define more than one transition at once. It is equivalent to several `addTransition` calls.

```
table = table.addTransitions(
    State.ACTIVE, {
        Input.ARM_UNLOCKED: ([], State.UNLOCKED),
        Input.ARM_LOCKED: ([], State.LOCKED),
        })
```

One thing to note here is that the outputs are `lists` of symbols from the output set. The output of any transition in Machinist is always a `list`. This simplifies the definition of output symbols in many cases and grants more flexibility in how a machine can react to an input. You can see one way in which this is useful already: the transitions out of the `ACTIVE` state have no useful outputs and so use an empty `list`. The handling of these `lists` of outputs is discussed in more detail in the next section, Output Executors.

3.3 Output Executors

The last thing that must be defined in order to create any state machine using Machinist is an *output executor*. In the previous sections we saw how the outputs of a state machine must be defined and how transitions must specify the outputs of each transition. The outputs that have been defined so far are only symbols: they can't have any impact on the world. This makes them somewhat useless until they are combined with code that knows how to turn an output symbol into an **actual** output. This is the output executor's job. Machinist provides a helper for writing classes that turn output symbols into side-effects:

```
from machinist import MethodSuffixOutputer

class Outputer(object):
    def output_ENGAGE_LOCK(self, engage):
        print("Engaging the lock.")

    def output_DISENGAGE_LOCK(self, disengage):
        print("Disengaging the lock.")

outputer = MethodSuffixOutputer(Outputer())
```

When used as the output executor for a state machine, the methods of this instance will be called according to the names of the outputs that are produced. That is, when a transition is executed which has `Output.ENGAGE_LOCK` as an output, `output_ENGAGE_LOCK` will be called. This lets the application define arbitrary side-effects to associate with outputs. In this well-defined way the otherwise rigid, structured, explicit state machine can interact with the messy world.

3.4 Construction

Having defined these things, we can now use `constructFiniteStateMachine` to construct the finite state machine.

```
from machinist import constructFiniteStateMachine

turnstile = constructFiniteStateMachine(
    inputs=Input,
    outputs=Output,
    states=State,
    table=table,
    initial=State.LOCKED,
    richInputs=[],
    inputContext={},
    world=outputer,
)
```

Apart from the inputs, outputs, states, transition table, and output executor, the only other argument to pay attention to in this call right now is *initial*. This defines the state that the state machine is in immediately after `constructFiniteStateMachine` returns.

3.5 Receiving Inputs

Having created a state machine, we can now deliver inputs to it. The simplest way to do this is to pass input symbols to the `receive` method:

```
def cycle():
    turnstile.receive(Input.FARE_PAID)
    turnstile.receive(Input.ARM_UNLOCKED)
    turnstile.receive(Input.ARM_TURNED)
    turnstile.receive(Input.ARM_LOCKED)
```

If we *combine all of these snippets* and call `cycle` the result is a program that produces this result:

```
Disengaging the lock.
Engaging the lock.
```

CHAPTER 4

Examples

4.1 Model a Turnstile

```
# This example is marked up for piecewise inclusion in basics.rst.  All code
# relevant to machinist must fall between inclusion markers (so, for example,
# __future__ imports may be outside such markers; also this is required by
# Python syntax).  If you damage the markers the documentation will silently
# break.  So try not to do that.

from __future__ import print_function

# begin setup
from twisted.python.constants import Names, NamedConstant

class Input(Names):
    FARE_PAID = NamedConstant()
    ARM_UNLOCKED = NamedConstant()
    ARM_TURNED = NamedConstant()
    ARM_LOCKED = NamedConstant()

class Output(Names):
    ENGAGE_LOCK = NamedConstant()
    DISENGAGE_LOCK = NamedConstant()

class State(Names):
    LOCKED = NamedConstant()
    UNLOCKED = NamedConstant()
    ACTIVE = NamedConstant()
# end setup

# begin table def
from machinist import TransitionTable

table = TransitionTable()
# end table def

# begin first transition
table = table.addTransition(
    State.LOCKED, Input.FARE_PAID, [Output.DISENGAGE_LOCK], State.ACTIVE)
# end first transition

# begin second transition
```

```
table = table.addTransition(
    State.UNLOCKED, Input.ARM_TURNED, [Output.ENGAGE_LOCK], State.ACTIVE)
# end second transition

# begin last transitions
table = table.addTransitions(
    State.ACTIVE, {
        Input.ARM_UNLOCKED: ([], State.UNLOCKED),
        Input.ARM_LOCKED: ([], State.LOCKED),
        })
# end last transitions

# begin outputer
from machinist import MethodSuffixOutputer

class Outputer(object):
    def output_ENGAGE_LOCK(self, engage):
        print("Engaging the lock.")

    def output_DISENGAGE_LOCK(self, disengage):
        print("Disengaging the lock.")

outputer = MethodSuffixOutputer(Outputer())
# end outputer

# begin construct
from machinist import constructFiniteStateMachine

turnstile = constructFiniteStateMachine(
    inputs=Input,
    outputs=Output,
    states=State,
    table=table,
    initial=State.LOCKED,
    richInputs=[],
    inputContext={},
    world=outputer,
)
# end construct

# begin inputs
def cycle():
    turnstile.receive(Input.FARE_PAID)
    turnstile.receive(Input.ARM_UNLOCKED)
    turnstile.receive(Input.ARM_TURNED)
    turnstile.receive(Input.ARM_LOCKED)
# end inputs

if __name__ == '__main__':
    cycle()
```

CHAPTER 5

Indices and tables

- *genindex*
- *modindex*
- *search*