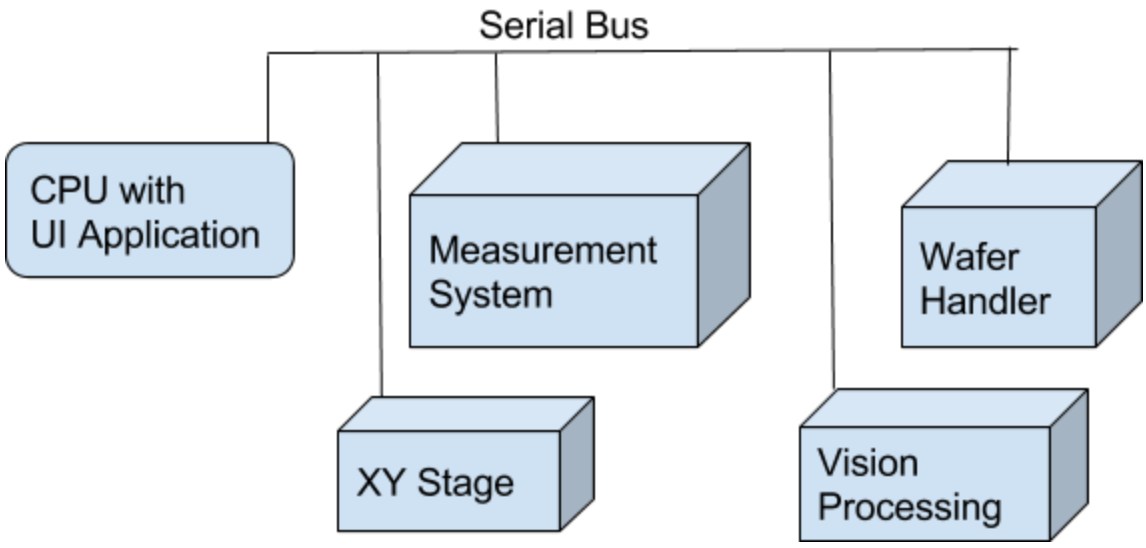
A structured approach to asynchronous state machines

Synopsis: Complex state machines are difficult to maintain in the face of rapidly changing requirements. Code from disparate parts of a system needs to be executed in each state, requiring engineers be fluent in all parts of the system. Additionally, changes to the state diagram need to be communicated and discussed across the entire project. The end result is that many people are modifying the same code which can lead to mistakes and frustration.

Example system--

Let's consider a wafer metrology tool that measures thickness of various films in a variety of methods.

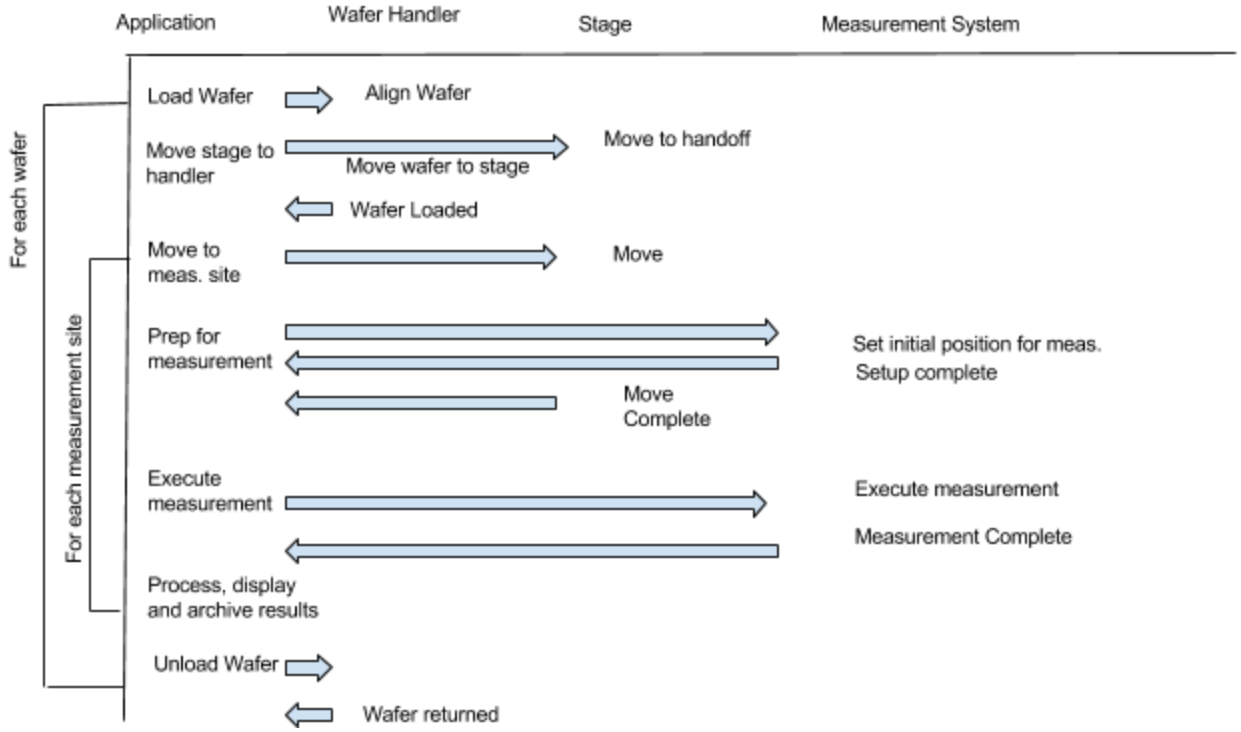


The system has an application that provides the user the ability to define measurements, train the system to find measurement locations, and define post measurement processing and result display. It also provides feedback during processing and diagnostic controls that interact with the tool at a finer granularity.

The system is designed for two user groups: an application engineer who understands the measurements and can define a measurement recipi and technician who is responsible for executing the recipe and verifying that the process completed successfully.

The execution model looks like:

Recipe Execution:



Some notes:

In this example, each device is controlled by a separate process and communication is handled through an OS supplied API for message queues and semaphores.

Measurement time and the number of measurements might vary greatly, altering whether wafer movement or measurements is the limiting factor for throughput.

Multiple, differing measurements might occur at a site requiring different setup, collection and processing.

Measurement setup may include a variety of controls: setting laser power, changing optics positions, focussing on the measurement site, turning sensors on/off, etc. These operations may generally be executed concurrently.

Traditional Solution:

A conventional approach would be to have a state machine that transitions for each action that needs to occur in the process. In practice, this works well for certain parts of the system: The process of loading, aligning and unloading wafers rarely changes and the operations may be reliably overlapped without concern of frequent change.

The measurements the system provides may be under continuous change due to customer requirements, improved measurement methodologies that affect the hardware used, and manufacturing concerns and constraints. Continuously updating measurement state machines poses a couple of challenges:

It’s difficult to revert to a previous configuration once a change is implemented.

Experience with all of the system mechanisms as well as the flow of the state machine is

required to make modifications.

The easiest approach is to replicate the entire state machine when a new application is being developed which results in duplicate code.

Consider a measurement with the following components:

An initial setup that requires:
- laser power to be set and verified
- A laser safety shutter opened
- A filter to be moved into position

A measurement that includes:
- The laser to be set at three separate power levels
- The sensor gain be adjusted for each power level
- Data retrieved and processed

If each action happened instantaneously, it would be easy to write straight line functional code, eg:

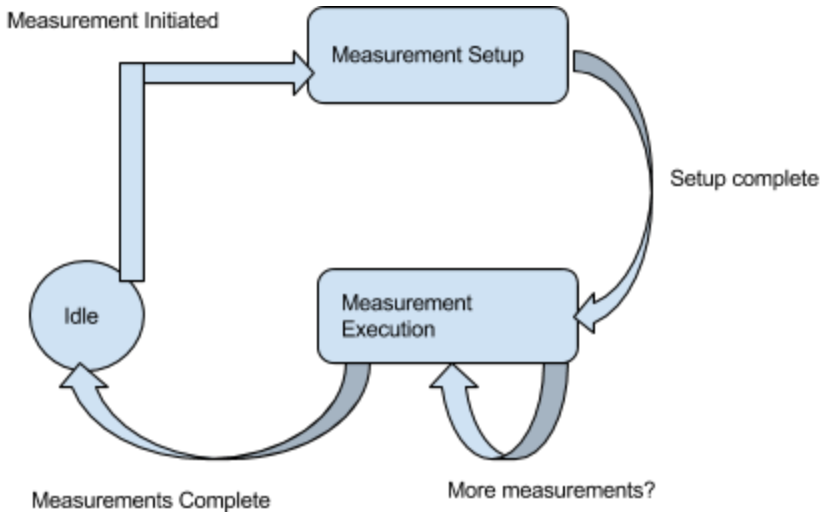
```
function measure(recipe: meas_settings)
{
   setLaserPower(meas_settings.power[0])
   setShutter(meas_settings.shutter_position);
   setFilter(meas_settings.filter_position);
   raw_data = array(float);
   foreach power in  meas_settings.power {
       setLaserPower(power);
       setSensorGain(calcGain(power));
       raw_data += readSensor();
    }
    return processData(raw_data);
}
```

But in reality we want to overlap the operations that can be overlapped. Let's also use a more OO approach:

```
function measure(recipe: meas_settings)
{
   laser = new Laser();
   optics = new Optics();
   sensor = new Sensor();
//       set up for the measurement
   laser.setPower(meas_settings.power[0])   // non-blocking, command sent to controlling
process
   optics.Shutter(open);
```

```
    while (!optics.idle());       // wait for operation to complete
    optics.Filter(open);
    while (!(optics.idle() && laser.idle());    // wait for operations to comple
    raw_data = array(float);
    foreach power in meas_settings.laserPower {
      laser.setPower(power);
      sensor.setGain(sensor.calcGain(power));
      while(! (laser.idle() && sensor.idle());   // wait
      raw_data += sensor.Read();    // blocking call, assumed to be reasonably quick
     }
     return processData(meas_settings.algorithm, raw_data);
}
```

This is already getting to the point where a state machine would make more sense (simplified here):



But projects evolve and invariably become more complicated. Imagine that the requirement becomes:

Move to a site
Take a series of measurements around the site
Return to the primary site and take measurements in multiple configurations

Although the logic is still relatively simple, an implementation that is concurrent while maintaining code reusability isn't well suited for a state machine approach.

An alternative is to have a factory that constructs a measurement tree that is then navigated via a visitor.

Eg:

```
Class MeasurementBase() {
   Point measurementLocation;
   MeasurementBase children[];
   accept(visitor v);
}

Class MunctionDepth::MeasurementBase() {
     OpticSetup opticSettings;
     LaserSetup laserSettings[];
     SensorSetup sensorSettings[];
     MeasurementCalibration cal;
     accept(visitor v);
}

Class mobility:measurementBase() {
     OpticSetupB opticSettings;          // A different (non-dynamic) hardware configuration
     LaserSetup laserSettings;           //
     accept(visitor v);
}

Class stageMove : measurementBase() {
      Point locations[];
      accept(visitor v);
}

Class visitor {
    Laser las;
    Optics optics;
    Sensor sensor;
    MeasObserver observer[];
    registerObserver(MeasObserver ob);
    Result results[];
    visitElement(LaserSetup las);
    visitElement(Point points[]);
    visitElement(OpticsSetup optics);
}
```

The visitor class is tasked with understanding the parallel operations that may be performed for a given hardware configuration and based on which elements are present in a given setup can decide which operations to overlap.

Instead of data being processed explicitly at each step, the measurement results are reported data to the UI and to GEM/SECS stream via observer and the full set of results is available.